Ph.D. DISSERTATION

Last-level Cache Partitioning through Memory Virtual Channels

메모리 가상 채널을 통한 라스트 레벨 캐시 파티셔닝

BY

Chung Jongwook

FEBRUARY 2023

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Last-level Cache Partitioning through Memory Virtual Channels

메모리 가상 채널을 통한 라스트 레벨 캐시 파티셔닝

BY

Chung Jongwook

FEBRUARY 2023

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Last-level Cache Partitioning through Memory Virtual Channels

메모리 가상 채널을 통한 라스트 레벨 캐시 파티셔닝

지도교수 김 장 우

이 논문을 공학박사 학위논문으로 제출함

2023년 1월

서울대학교 대학원

전기 · 정보 공학부

정 종 욱

정종욱의 공학박사 학위 논문을 인준함

2023년 1월

위 원 장:	김 재 준	(인)
부위원장:	김 장 우	(인)
위 원:	심 재 웅	(인)
위 원:	이 병 영	(인)
위 원:	김 한 준	(인)

Abstract

Last-level Cache Partitioning through Memory Virtual Channels

Jongwook Chung
Department of Electrical and Computer Engineering
The Graduate School
Seoul National University

Ensuring fairness or providing isolation between multiple workloads with distinct characteristics that are collocated on a single, shared-memory system is a challenge. Recent multicore processors provide last-level cache (LLC) hardware partitioning to provide hardware support for isolation, with the cache partitioning often specified by the user. While more LLC capacity often results in higher performance, in this dissertation we identify that a workload allocated more LLC capacity result in worse performance on real-machine experiments, which we refer to as MiW (more is worse). Through various controlled experiments, we identify that another workload with less LLC capacity causes more frequent LLC misses. The workload stresses the main memory system shared by both workloads and degrades the performance of the former workload even if LLC partitioning is used (a balloon effect).

To resolve this problem, we propose virtualizing the data path of main memory controllers and dedicating the memory virtual

channels (mVCs) to each group of applications, grouped for LLC partitioning. mVC can further fine-tune the performance of groups by differentiating buffer sizes among mVCs. It can reduce the total system cost by executing latency-critical and throughput-oriented workloads together on shared machines, of which performance criteria can be achieved only on dedicated machines if mVCs are not supported. Experiments on a simulated chip multiprocessor show that our proposals effectively eliminate the MiW phenomenon, hence providing additional opportunities for workload consolidation in a datacenter. Our case study demonstrates potential savings of machine count by 21.8% with mVC, which would otherwise violate a service level objective (SLO).

Keywords : Cache Partitioning, Main Memory, Memory Virtual Channel, More is Worse, Fairness, QoS
Student Number : 2014-21730

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern chip multiprocessors (CMPs) consist of multiple cores sharing various resources, including shared last level cache (LLC), on-chip interconnect, and main memory [6], [32], [49]. CMPs are currently the most popular design choice for servers used in cloud environments, and such CMP-based servers consistently run several heterogeneous applications to satisfy the needs of diverse users. This trend is becoming more prevalent with the emergence of virtual machines and containers for cloud services.

When applications run simultaneously, contention and interference of shared resources in a system can cause performance degradation for some or all the applications [9], [18], [30], [39], [47], [48], [49]. As a result, there has been a significant amount of prior work done to provide fairness and

minimize interference from sharing the on-chip LLC capacity and main memory bandwidth [13], [39], [47], [48], [49]. In particular, when multiple applications compete for a limited capacity of shared cache, high-priority applications that need quality-of-service (QoS) guaranteed, or real-time applications can suffer from performance degradation due to excessive cache occupancy from other applications [6], [18], [32], [42]. To ensure the performance guarantee for QoS or real-time applications, modern CMPs provide cache partitioning (CP) [3], [8], [15] where different portions of LLC are allocated to different applications. Cache partitioning can allocate an isolated cache region to high-priority applications, which avoids contention and interference by preventing concurrently running applications from evicting high-priority application cache lines [15]. Many prior studies [47], [48] have investigated alternative CP to maximize overall performance. However, recently, CMPs [15] provide user-specified CP, and the previously proposed CP algorithms are not necessarily applicable. In this dissertation, we propose a mechanism to enforce performance isolation in user-specified LLC partitioning.

When a CMP dedicates more LLC capacity to a process group through cache partitioning, the intuitive expectation is that performance improves [15]. However, we demonstrate that the opposite can occur as a process group can actually perform worse when it obtains more LLC capacity. We refer to this as more-is-worse (MiW) phenomenon and define MiW degree as the ratio of

Instructions Per Cycle (IPC) when maximum LLC is allocated to a process group to the maximum IPC that can be obtained through CP. Evaluations show that MiW degree can reach up to 39.5% with synthetic workloads, 14.4% or SPEC CPU2006 [45], and 547.0% for TailBench [20] benchmarks, respectively, on Intel Broadwell-based [23] Xeon systems.

In this dissertation, we first provide an analysis of why this MiW phenomenon occurs. When a particular process (e.g., process A) receives more LLC capacity, another process in the system (e.g., process B) comes to receive a smaller fraction of the LLC capacity and experiences higher LLC Miss Per Kilo Instructions (MPKI). This increases main memory bandwidth demand from process B (a balloon effect[①]) and results in higher main memory access latency for all the processes. Even though the memory access patterns for process A and process B are different (i.e., accessing different banks or ranks), both processes share the same data path to the main memory system, including memory request buffers. As a result, requests from process B can monopolize the shared data path resource in the memory system. This effectively results in process B "blocking" process A's requests and degrades the performance of process A.

To prevent this blocking in the data path to the main memory, we propose to virtualize the data path of memory controllers with

[①] We use the terminology balloon effect since changes in one area (i.e., cache partitioning) leads to an adverse effect in another area (i.e., memory bandwidth).

memory virtual channel (mVC) where a separate memory request buffer is provided for each group of LLC. The overall memory request buffer storage is partitioned across the number of groups supported in the LLC, which is equivalent to the number of mVCs. DRAM commands from different buffers (or mVCs) are arbitrated and served independently - thus, each mVC has effectively a private data path to the memory channel and avoids blocking. The memory controller requires a mVC arbiter that is responsible for arbitrating between the mVCs - the mVC that receives a grant from the mVC arbiter gains access to the memory channel. The grant is released only after a column-level DRAM (RD/WR) command is issued to avoid unnecessary DRAM row-buffer conflicts.

We discuss mVCs with four different buffer allocation policies, which are static, proportional, inversely-proportional (both based on its share of LLC ways), and dynamic partition. The observations show that static and proportional partitions are more effective in eliminating MiW than the other. Furthermore, we explore the design space by observing the performance of mVC on various ratios of buffer allocation. As a result, we show that it is possible to select an appropriate configuration satisfying the target performance for the group with more LLC capacity, and also maximizing the performance of the group with less LLC capacity. Our case study shows that when satisfying 90% of the standalone performance, with mVCs we can save 21.8% of machines by sharing the machines among applications in a distributed system.

1.1 Research Contributions

In this dissertation, we make the following contributions:

- This is one of the first work to demonstrate the problem of MiW (more-is-worse) on a real machine, where allocating more LLC capacity to a workload leads to worse performance due to an increased degree of congestion (blocking) on the main memory shared by all the workloads (a balloon effect).
- We propose to virtualize the data path of memory controllers to mitigate this blocking problem and explore the design space of memory request buffer allocation.
- We evaluate memory virtual channels (mVC) using a cycle-level simulator, which effectively eliminates MiW and recovers lost IPC due to the blocking.

  We perform a case study to demonstrate mVC can provide additional opportunities for workload consolidation to save the machine count by up to 21.8%, which would otherwise violate a service level objective (SLO)

1.2 Outline

The organization of this dissertation is as follows.

In Chapter 2, we explain modern CMP system's cache hierarchy and cache partitioning. Chapter 3 describes the More-is-Worse phenomenon. We show MiW with real machines and the impact on latency-critical workloads. We also explore the root cause of MiW. In Chapter 4, to overcome MiW, we propose memory virtual channel (mVC) with several buffer allocation strategies. We quantify the benefits of mVC when applied to main memory system, compared with the conventional system. Related works are presented in Chapter 5. Finally, Chapter 6 presents the conclusion of this dissertation.

Chapter 2

Background

In this chapter, we describe the memory hierarchy in modern computers. Cache allocation can be used to assign the shared LLCs to each CPU core and is discussed in detail in this chapter. Also, analyzes of the benchmark used in this paper are included in this chapter.

2.1 Memory Hierarchy and Cache Policies

Modern computer system is a configuration of multiple CPU cores connected to a memory hierarchy. CPU cores must fetch data and instructions from the cache memory for performance reasons.

However, cache memory is an expensive resource, so computer systems leverage memory hierarchies of cheaper and slower memories but larger capacity, such as DRAM. The memory hierarchy effectively hides and reduces the memory access time by storing data and instructions in a cache, considering special and temporal locality.

Modern CMPs include multiple levels of cache. For example, Intel' s Xeon processors include three levels of cache which are level 1 (L1), level 2 (L2), and level 3 (L3, also known as the last level cache (LLC)) caches. The lowest level cache, L1 cache, The L1 cache is located closest to the CPU and is the fastest but has the smallest capacity. As the level goes up to L2 and LLC, the capacity increases, but the speed decreases. If data exists in the L1 cache (cache hit), the CPU uses the data immediately, otherwise (cache miss), the CPU checks the data at higher levels of the cache and memory hierarchy. If the data is not available in LLC, CPU requests the data from the main memory. Different cache replacement policies may be implemented to evict cache lines to make space for subsequent requests [58].

The biggest difference between L1, L2 caches and LLC is that L1 and L2 caches are independent for each core, but LLC is shared between cores. Each physical core contains its own private L1 and L2 caches, but LLC is shared and can be fully accessed and utilized by all of the cores in the system. Having all the cores share the LLC has the advantage that cores that need a lot of cache space can

access more of the cache because space is not wasted on cores with low LLC utilization, resulting in higher cache utilization for the entire system.

CMP runs multiple threads in parallel. As a result, the contents of the shared LLC can be quickly overwritten with new data requested by the core from memory [59]. This situation is highly dependent on the number of concurrently running threads and their respective memory access patterns. Under moderate workloads and conditions, a large portion of the LLC can be overwritten with new data read from memory, which evicts significant data of L1 and L2 caches, reducing the performance of those cores [56], [57].

For example, consider a situation where CMP is running several processes and workloads concurrently that have low priority but generate a very large amount of memory traffic. Also assume that the CMP has an Interrupt Service Routine (ISR) programmed to handle latency sensitive high-priority interrupts. In this situation, low-priority processes between interrupts will generate a lot of memory traffic, overwriting the entire LLC with new data read from memory, thus invalidating the data in the L1 and L2 caches of the cores. If a high-priority interrupt occurs at this time, the interrupt will experience very high latency. This is because the code and data to handle the interrupt are no longer in the cache and need to be fetched from memory again.

2.2 Cache Partitioning

To overcome the contention and interference on the shared resources, CMPs provide cache partitioning/allocation techniques [3], [15], [60]. Cache partitioning (CP) divides shared LLC resource and dedicates each partitioned LLC to a group (class) of processes (Figure 2.1).

CP allows the cache to be adequately allocated according to the working set size or cache sensitivity of a process group, alleviating contention, and interference between processes [39], [47]. For example, AMD provides CP in Opteron [3], [8], and Intel introduced Cache Allocation Technology (CAT) starting from Haswell architecture [15].

CP techniques can be classified as way, set, or block (line) based partitioning (Figure 2.2) [1], [7], [32], [33], [41], [42], [53]. Way-based partitioning [7] divides LLC by cache ways. Processes can replace the cache line only within the allocated cache ways.

Way-based partitioning is relatively cheap to implement because the process can access all the cache sets regardless of the number of allocated ways. However, it is limited to the maximum number of ways in granularity, and the associativity of each partition can be greatly reduced depending on the allocated ways [42].

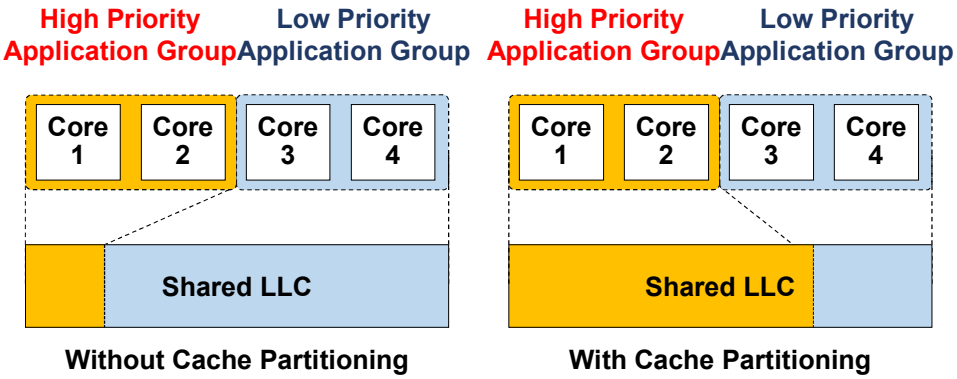


Figure 2.1: Without cache partitioning, low priority application group can occupy almost all the shared LLC which can lead to performance degradation to high priority application groups. Cache partitioning can dedicate shared LLC to high priority application groups.

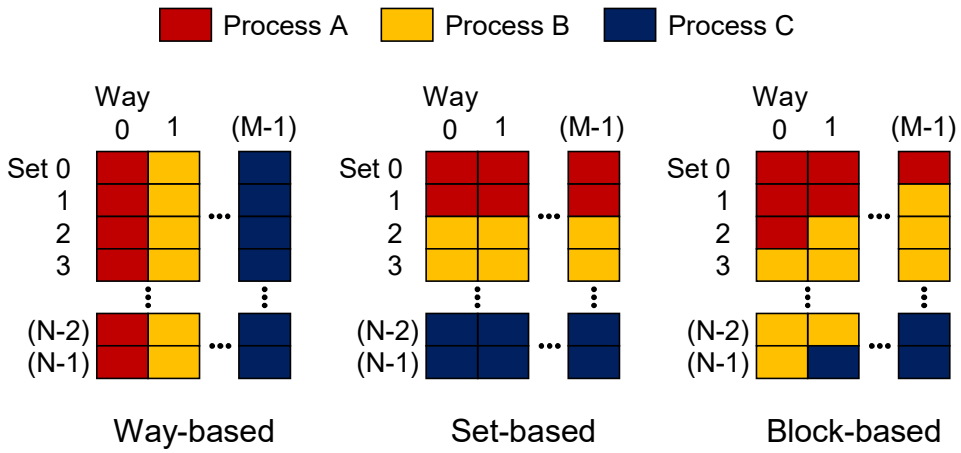


Figure 2.2: Three strategies of cache partitioning. Cache partitioning can be classified as way, set, or block (line) based partitioning.

Way-based partitioning is relatively cheap to implement because the process can access all the cache sets regardless of the number of allocated ways. However, it is limited to the maximum number of ways in granularity, and the associativity of each partition can be greatly reduced depending on the allocated ways [42].

Set-based partitioning [1], [33] (or page coloring [53]) partitions LLC by sets instead of ways, and each process gets several sets from the cache. LLC is virtually divided so that the address of a requested data is mapped to a set in the virtual cache. The virtual set index is then mapped to the actual physical cache set index. This translation makes set-based partitioning more expensive than way-based partitioning, especially when resizing the partition.

For finer-grained partitioning, block-based partitioning was also proposed to partition the cache-by-cache block (line) granularity [41] and provide more cache partitions. However, the complexity and overhead for managing and storing the mapping information identifying the owner of each cache line are high [32].

AMD Opteron [3] implements set-based cache partitioning. To minimize the amount of LLC data being evicted by a core that does not allocate the data, the Opteron processor can direct L2 victim traffic to a specified set of the LLC. However, the unit of partitioning is a quarter of the total LLC capacity, which is too coarse-grained.

By contrast, Intel CAT [15] adopts way-based CP for the shared LLC. With Intel CAT, each class of service (CLOS) consists of one or more applications. A bitmask (each bit representing a single cache way) is used to determine the amount of LLC allocation for each CLOS, and the bitmask can be changed dynamically at runtime. CLOS can be allocated exclusively (isolated mode) or allocated to overlap with other CLOS (overlapped mode). CAT has been supported since Haswell microarchitecture with 4 CLOSs; more recent Broadwell and Skylake-based servers support up to 16 CLOSs. In Intel CAT, CP can be managed with a program called pqos. One or more cores can be dedicated within a CLOS. The default CLOS is CLOS 0, and at first, all of the cores are dedicated with CLOS 0. To allocate some portion of the shared LLC to a CLOS, available cache ways are specified through the capacity bit masks.

There are multiple prior works on CP to limit the impact of contention and interference on the shared LLC. While most of these work concern limiting a low-prioritized workload from interfering with a prioritized application, not many of them study having prioritized applications competing for the shared LLC.

P. Veitch et al. [56] describes an approach for mitigating the effects of a noisy neighbor (which is a low priority application but occupying most of the LLC) has on the LLC in a system by using CP. CP is used to allocate less LLC capacity to the low priority applications with the result that more data from other high priority

applications can remain in the shared LLC as it is not evicted by low priority applications. Also, CP can be used to make performance more predictable and deterministic, by mitigating noisy neighbor effects on CMP resources such as LLC. According to the results of the study, the average latency reduced between 47% and 92% when comparing the case where LLC was equally allocated to processes using CP and the case where noisy neighbors occupied more LLC than other processes because CP was not used.

Intel studied CP to limit the impact of noisy neighbor applications on the performance of other applications [57]. In this study, STREAM benchmark was used as a noisy low priority application, and bzip2 was used as a high priority application. The STREAM benchmark is suitable for noisy priority applications as it over-utilizes LLC due to its high memory usage. When running bzip2 and STREAM at the same time, the execution time of bzip2 greatly increased. This is because the LLC share of bzip2 was very small compared to STREAM. When the LLC share of STREAM was limited using CP, the LLC share of bzip2 increased and the execution time significantly decreased. This can be taken as an example in which CP limits the LLC occupancy of noisy low priority applications, thereby reducing LLC contention and improving the performance of high priority applications.

Herdrich et al. [15] demonstrated the performance improvement of up to 4.5× from CAT when running SPEC CPU2006 applications together as CAT significantly alleviated the

performance degradation of an application from interference. With CP (e.g., CAT), more LLC capacity can be dedicated to a certain application to prioritize and improve its performance effectively. However, contrary to this intuitive expectation, we observed that a group of processes could perform worse when they receive more LLC capacity.

2.3 Benchmarks

Many research results in the field of computer architecture are measured and reported through benchmarks. These benchmarks represent current or future software in specific application areas. Therefore, benchmark suites are provided by various corporations, research organizations, communities, or companies. Representative benchmarks include SPEC [45], Cloud Suite, and Tail Bench [20], each targeting general purpose computing, cloud computing, and real time processing. In this dissertation, SPEC and Tail Bench were used to focus on general purpose computing and real time applications.

2.3.1 Working Set Size

We first focus on the working set size of SPEC2006. The working set size of an application is the estimate memory actually used by an active application [61]. Representative metrics for measuring the working set size are virtual size (VSZ) and resident set size (RSS).

VSZ is the memory address space reserved by the OS for the application and is the total memory usage of the application. This memory address space is used to hold data or instructions, and not all of them are physically stored in memory.

RSS represents the amount of physical memory actually used by the application. In general, RSS is equal to or less than VSZ. For example, if the system's physical memory is insufficient, a portion of the memory used by the application is paged out to disk. In this case, RSS decreases but VSZ does not change. Also, the application does not require as much physical memory as VSZ to run [61], [62].

Previous studies have shown that RSS and VSZ of CPU2006 are used up to 1GB [61], [62]. However, these results of experiments conducted in an old architecture and small memory environment.

2.3.2 Top-down Analysis

A top-down analysis has been done on SPEC CPU2006 benchmarks [63]. This top-down analysis is a practical method to identify bottlenecks in out-of-order processors. The study was conducted on Intel 3rd generation (codenamed Ivy Bridge) and used Intel VTune [64] and standard Linux perf [65] utility tools. The top-down analysis breakdown shows the top level, backend level, and memory level breakdown in single-thread mode and multi-core mode. We will use the results to analyze our results. Similar top-down analysis has been done on SPEC CPU2017 [66], [67].

In the top-down analysis we focus on the backend bound category. Backend bounds reflect slots where uops are not delivered in the problem pipeline because the backend lacks the resources needed to accept the uop. An example of a problem in this category is a data cache miss or hang due to an overloaded divider.

Backend bound is divided into memory bound and core bound. This is achieved by granularizing backend outages based on the occupancy of execution units in every cycle. Naturally, you need to keep the execution unit busy to maintain maximum IPC. For example, on a 4 wide machine, if some code has less than 3 uops running at steady state, you won't achieve an optimal IPC of 4. These suboptimal cycles are called execution delays.

Memory bound corresponds to the execution delay associated with the memory subsystem. These hangs usually show up as execution units starving after a short period of time, such as in the case of a load that misses all caches.

Core bound outages can manifest as brief execution starvation periods or suboptimal execution port utilization. While high-latency divide operations can serialize execution, pressure on execution ports serving certain types of uops can result in fewer ports being used in cycles. Core bound problems can often be mitigated by better code generation.

As mentioned in chapter 2.1, modern CPUs implement multiple levels of cache hierarchy to hide the latency of external memory. To determine the real penalty for memory access, the true penalty for memory access is when the scheduler is not prepared to feed execution units. For example, L1D caches often have low latencies comparable to ALU delays. However, in certain scenarios, such as loads blocked for forwarding data from old storage to overlapping addresses, you may experience high latency while the load is eventually satisfied by the L1D cache. In these scenarios, the executing load lasts for a long time without any L1D cache misses. Therefore, it is tagged under the L1 Bound.

Store operations are buffered and executed after retirement on the out-of-order CPU due to the memory ordering requirements of the x86 architecture. In most cases, the performance impact is

small. However, it cannot be completely ignored. The store bound metric is defined top-down because the execution port utilization is low, and the store count is buffered for a portion of the cycle. If both load and store issues apply, prioritize load nodes based on the mentioned insights.

Data TLB misses can be classified as memory bound sub nodes. For example, if a TLB translation is satisfied by L1D cache, it will be tagged under the L1 Bound.

For memory bandwidth and memory latency bound, the occupancy of requests waiting for data to return from the memory controller is measured. Whenever the occupancy exceeds a certain threshold (for example, 70% of the maximum number of requests the memory controller can handle concurrently), it marks it as being potentially limited by the memory bandwidth. The rest is due to memory latency.

2.3.3 Profiling Tools

The measurement-based study conducted in this dissertation uses the SPEC utility to report execution times and SPEC CPU composite performance metrics. A set of modern tools for event-based sampling and profiling are also used, including the Intel performance counter monitor, and the Intel VTune Amplifier. These tools interface and collect information from the on-chip performance

monitoring unit (PMU), which is part of the fabric of modern processors. Statistics gathered from the PMU registers during benchmark execution include the number of clock cycles, the number of instructions executed, as well as numerous microarchitectural-specific events that capture the behavior of the processor's front-end and back-end resources. such as branch predictors, functional units, and memory hierarchies.

Chapter 3

More-is-Worse Phenomenon

We first demonstrate and analyze how the performance of a process group decreases as we allocate more LLC capacity with cache partitioning on real machines. To the best of our knowledge, this un-intuitive phenomenon has not been reported on real machines[②].

3.1 More LLC Leading to Performance Drop

We evaluated a system with a single socket Intel Xeon Broadwell server with 20 cores (40 hardware threads with Hyper-Threading), 50MB of shared LLC, and 76.8GB/s of peak main memory bandwidth.

[②] We used the isolated mode because the overlapped mode can cause unnecessary contention between the benchmarks on LLC, making the analysis more complicated.

Hardware Information	Setting Values
CPU Model	Intel Xeon E5-2698 v4
CPU Clock	2.2GHz
# of cores	20
# of memory controllers per CPU	2
Per Core:	
L1 I/D $ type/size/associativity	Private/32KB/8
L2 $ type/size/associativity	Private/256KB/8
L3 $ type/size/associativity	Shared/2.5MB/20
# of hardware threads	2
Hardware prefetch	Off
Per DDR4-2400 memory controller:	
# of channels	2
# of ranks per channel	2
Bandwidth per channel	19.2GB/s

Table 3.1: Hardware setup used in Chapter 3. Intel machine was used for cache partitioning.

Details of the experimental setup are described in Table 3.1. The Intel machine has CAT (Cache Allocation Technology) for cache partitioning (CP). Our initial evaluation uses SPEC CPU2006 benchmarks [45] and executed SPEC rate of N, where N means running N instances (processes) of a benchmark simultaneously. We bundled the cores that execute the same benchmark into one CLOS (class of service).

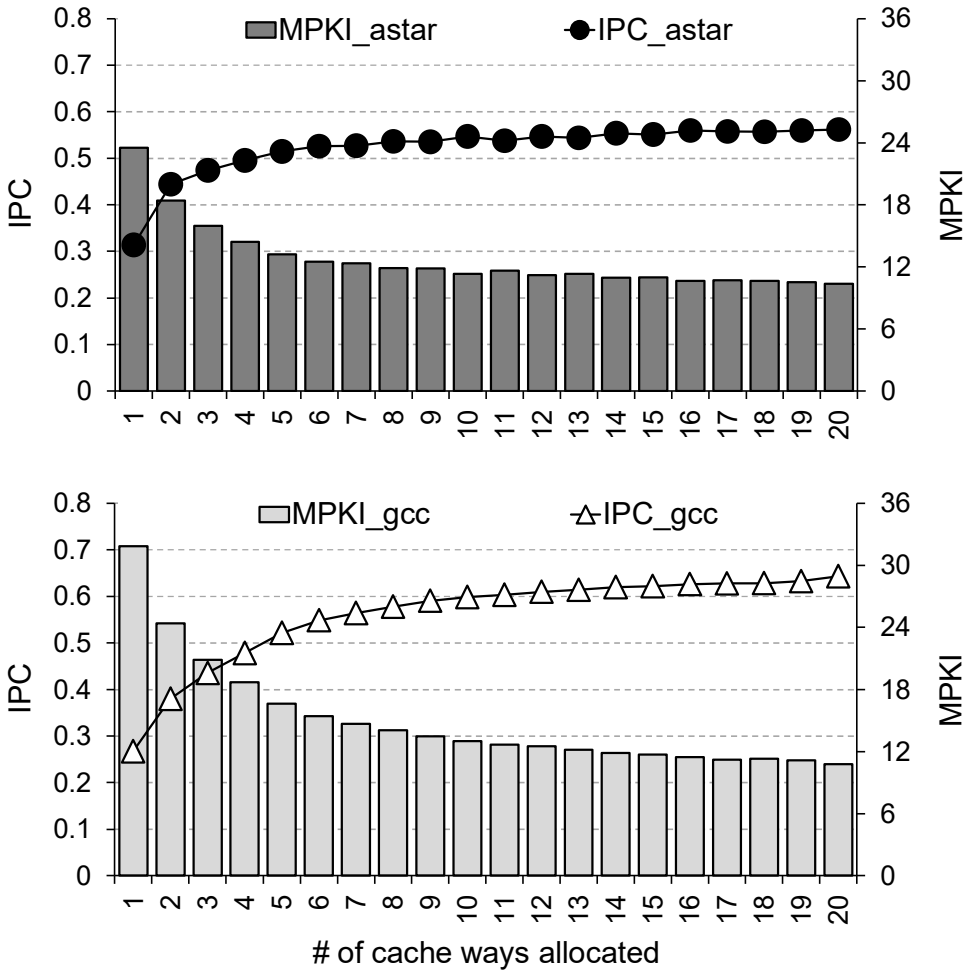


Figure 3.1: IPC and LLC Miss Per Kilo Instructions according to the change in the allocated LLC capacity for 473.astar and 403.gcc when executed alone respectively. IPC is normalized to when each run alone and occupies the entire LLC capacity (20 ways).

Figure 3.1 shows the IPC and LLC Misses Per Kilo Instructions (MPKI) variation as the number of allocated LLC ways increases when executing 473.astar and 403.gcc benchmarks alone with rate 20. Each core runs two instances; thus, we use 10 out of the 20 cores.

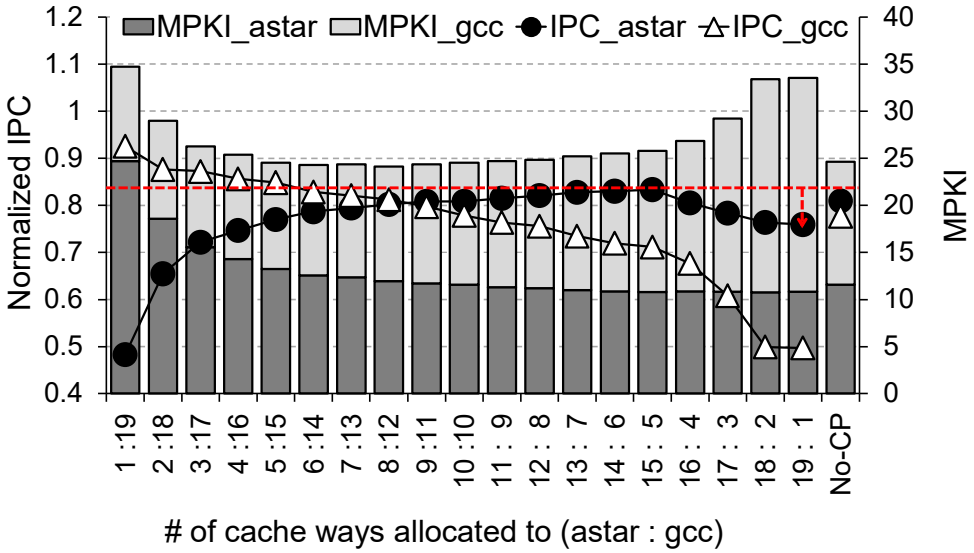


Figure 3.2: IPC and LLC MPKI according to the change in the allocated LLC capacity for 473.astar and 403.gcc when executed together. IPC is normalized to when each run alone and occupies the entire LLC capacity (20 ways). IPC of 473.astar decreases by up to 8.9% after reaching the peak when it is allocated with 15 LLC ways.

The evaluated Intel processor has 20 LLC ways per cache set, and thus, we swept the LLC ways from one to 20. The presented IPC is the mean IPCs from all the cores running the same application. The results are intuitive-as more LLC is allocated, MPKI decreases, and performance (IPC) monotonically increases. Initially, more LLC results in a significant decrease in MPKI and correspondingly a significant performance improvement but afterward, the change in MPKI is limited as performance saturates [39].

We then executed the two benchmarks together with each running on 10 physical cores and each with a rate of 20. We dedicated varying numbers of LLC ways to the two application groups: N to one and (20 - N) to the other. Figure 3.2 shows the normalized IPC and LLC MPKI when executing 473.astar and 403.gcc together, with the IPC and MPKI values of the two applications without CP in the rightmost column. Using CP improves the aggregate performance of the two application groups sharing the LLC. When we allocate nine LLC ways to 473.astar (11 for 403.gcc), its performance is the same as (2.7% better than) that without CP, showing CP is effective.

The expected behavior is a trade-off between LLC capacity and performance. As more LLC capacity is allocated to a workload, the performance is expected to continue to increase or saturate. However, our evaluation shows that performance can be degraded with more LLC capacity. For example, for 473.astar, performance first increases as LLC capacity increases, but beyond 15 LLC ways, the performance drops by up to 8.9%. This is seemingly counterintuitive as the performance of both 403.gcc and 473.astar are degraded when 473.astar occupies more than 15 LLC ways. We call this MiW (more-is-worse) phenomenon.

App A	App B	MiW
omnetpp	gcc	14.40%
astar	gcc	8.94%
sphinx	gcc	8.43%
gcc	gcc	6.01%
xz	xalancbmk	5.27%
mcf	blender	3.22%

Table 3.2: The degrees of MiW (more-is-worse) over pairs of applications (App A/B) which divide up LLC. The MiW degree is measured by comparing the aggregated IPC of App A when it occupies the maximum share of LLC (numerator) with the one when it performs best over all possible LLC shares (denominator) through CP.

In addition to 403.gcc and 473.astar, similar behaviors were also observed in other SPEC CPU2006 and SPEC CPU2017 [46] benchmarks. The degree of MiW, the ratio of IPC when maximum LLC is allocated to a process group to the maximum IPC that can be obtained through CP, for some of the SPEC benchmarks are summarized in Table 3.2. We observe up to 14.4% performance degradation when the former benchmark of the pair occupies more LLC capacity over a certain threshold, respectively. Note that MiW does not happen always. For example, on the pair of 473.astar-

473.astar, the performance of both groups increase monotonically as more LLC ways are allocated.

3.2 Synthetic Workload Evaluation

In this section, we evaluate the MiW phenomenon using synthetic workloads to better control workload's memory access characteristics and analyze performance degradation when allocating more LLC capacity. We use a pointer chasing synthetic workload, whose performance is sensitive to memory latency because of true dependency between each memory access. We controlled the degree of memory bandwidth pressure by varying the amount of data read per step of pointer chasing.

Without loss of generality, we call a group (class) of applications that are allocated more LLC capacity and expects higher performance 'group-A', and the other group that receives the remaining LLC capacity 'group-B'. To differentiate the characteristics of workload group-A and group-B, we set group-A to read only one cache line (64B) per pointer chasing step over 1GB of working set, which is 20× larger than the LLC capacity. Thus, group-A is less sensitive to changes in LLC capacity but more sensitive to changes in main memory access latency.

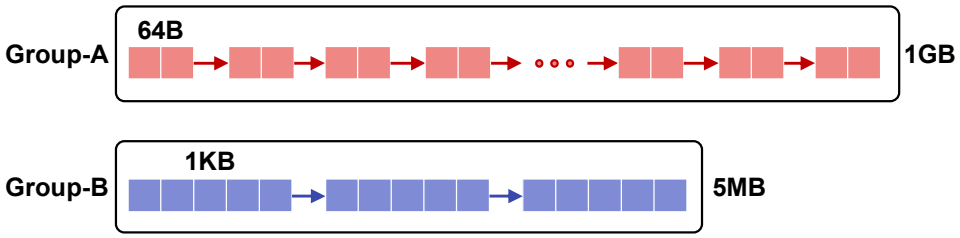


Figure 3.3: Synthetic workload used for evaluation of MiW. Group-A and group-B are pointer chasing workloads where group-A reads 64B of data in 1GB of working set, while group-B reads 1KB of data in 5MB of working set.

Group-B reads 1KB of data per pointer chasing step over 5MB of working set, which is only one-tenth of the system' s LLC capacity, to generate frequent LLC misses when smaller LLC capacity is allocated. We read 1KB of data per step to generate more bandwidth pressure to memory compared to group-A (Figure 3.3). We evaluated with the same system described earlier in Table 3.1, except only a single memory channel instead of four channels is used to stress main memory bandwidth.

Figure 3.4 shows the IPC and LLC MPKI as the number of LLC ways allocated to group-A and group-B is varied. For group-A that uses 1GB of memory and much larger than LLC capacity, its performance is mostly insensitive to the change in the allocated LLC capacity, and the memory bandwidth usage is maintained at a constant level of 1.8GB/s.

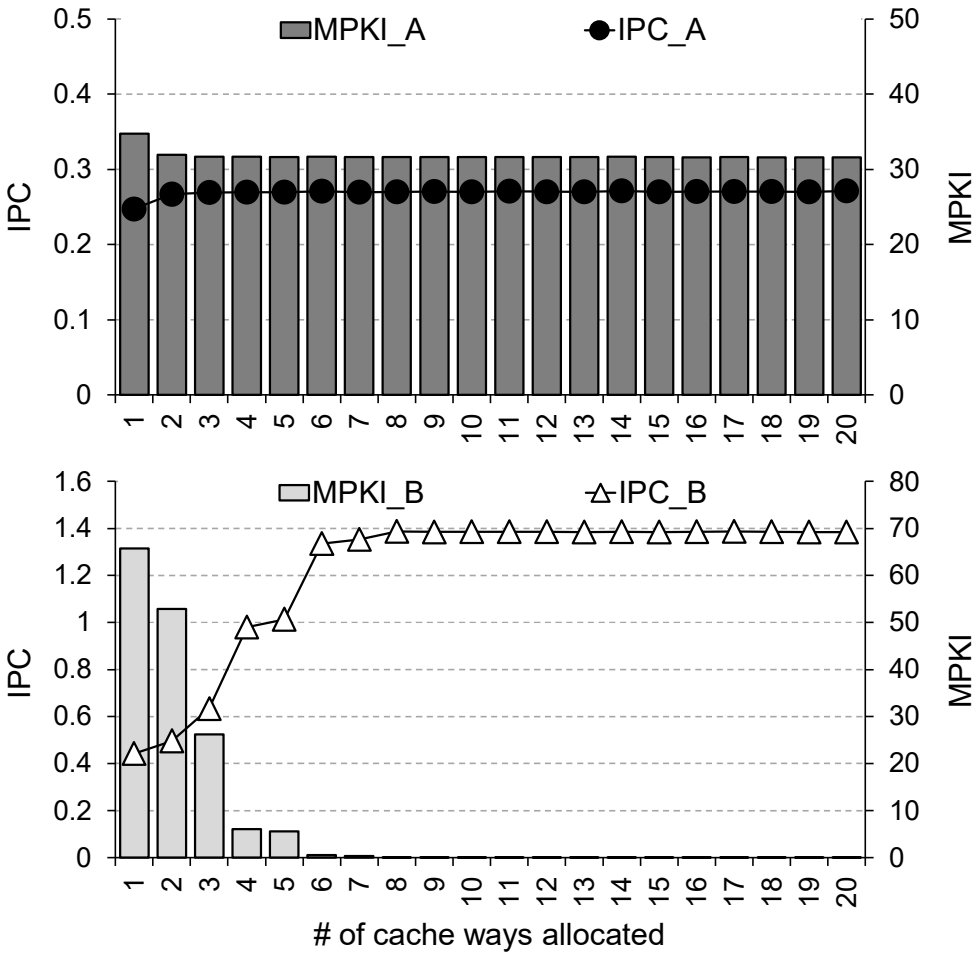


Figure 3.4: IPC and LLC MPKI of the synthetic workload when executing group-A and group-B alone respectively, with IPC normalized to when each workload runs alone with 20 LLC ways allocated.

By contrast, group-B uses only 5MB of memory and allocating a large amount of LLC capacity leads to negligible LLC misses. When the allocated LLC capacity is small LLC misses and memory access rates increase rapidly. Therefore, the IPC decreases by 68% and the memory bandwidth usage increases to 5.8GB/s.

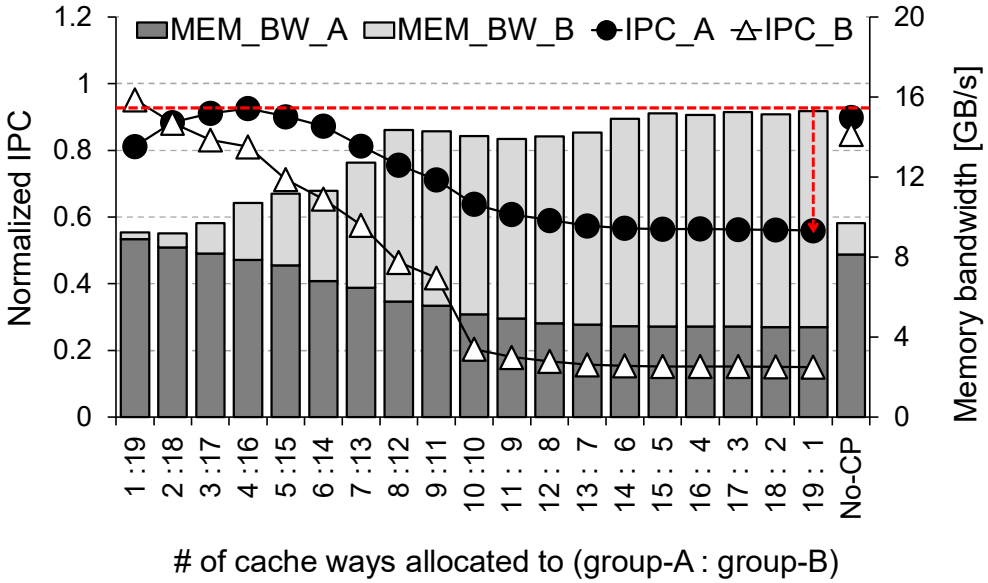


Figure 3.5: IPC and memory bandwidth of the synthetic workload when executing group-A and group-B together, with IPC normalized to when each workload runs alone with 20 LLC ways allocated. The IPC of group-A drops up to 39.5% after reaching the peak when it occupies 4 LLC ways.

The result when both group-A and group-B are executed is shown in Figure 3.5. When allocating more LLC capacity to group-A, we expect performance to increase or reach a steady-state, but performance decreases when 5 (25% of LLC capacity) or more LLC ways are allocated to group-A, reproducing MiW observed with SPEC benchmarks. Since group-A and group-B alone cannot fully utilize the system memory bandwidth, we executed group-A and group-B with rate four. The performance degradation (MiW) gets

worse as more instances of the group-A and group-B are populated. The synthetic evaluations demonstrate that MiW can be reproduced with a simple, synthetic workload but more interestingly, MiW can start even if an application group occupies only a smaller portion of the shared LLC resource.

3.3 Impact on Latency-critical Workloads

In addition to the SPEC benchmarks, we evaluate the impact of MiW on latency-critical (LC) workloads. It is well-known that LC applications, especially in datacenters, often require predictable and small tail latency [5], [10], [29]. However, as shown in Section III-A, MiW increases MPKI - thus, higher memory access latencies can significantly impact the tail latency problem [21]. Therefore, MiW can be even more critical for LC workloads.

To evaluate the impact of MiW on LC workloads, we used TailBench [20], [21] and executed each TailBench benchmark together with 403.gcc from SPEC CPU2006. Similar to the previous evaluations, we vary the number of LLC ways for the two benchmarks, but for the TailBench benchmarks, performance is measured in terms of tail latency. We used the single-node integrated configuration of TailBench, where a client and the corresponding LC application are integrated into a single process.

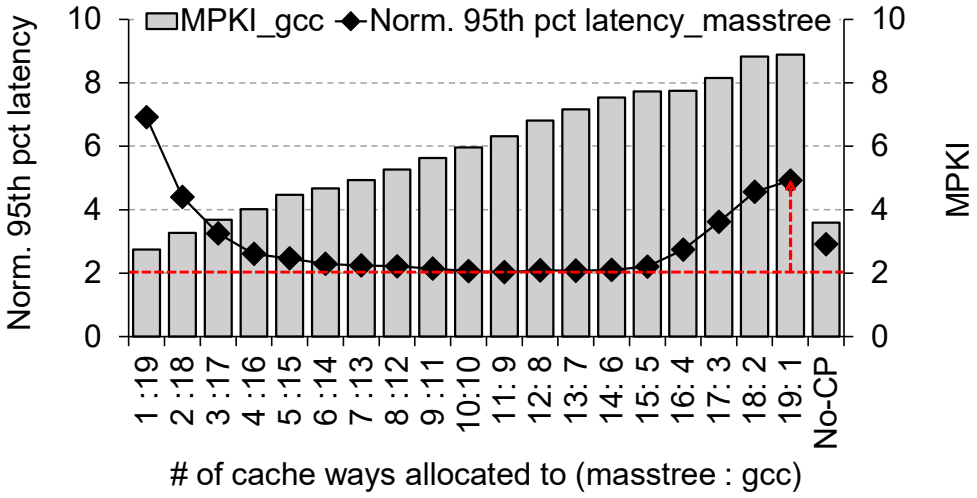


Figure 3.6: 95th percentile latency of masstree when executed with 403.gcc, where normalized to the tail latency executed alone occupying the entire LLC capacity. Similar trends were observed for other TailBench benchmarks.

Figure 3.6 shows the normalized 95th percentile latency of masstree, where normalized to the tail latency when runs alone occupying the entire LLC. The result shows that the tail latency increases by up to 143%, as it occupies more LLC ways. Table 3.3 summarizes the degree of MiW of other TailBench benchmarks. Moses and masstree have significantly higher MiW degrees compared to other benchmarks (as high as 547% with moses), due to higher LLC MPKI from these workloads, thus, results in longer queuing time. Due to space constraints, additional results are not shown, but similar trends were observed in evaluation of Intel Skylake machines, with tail latency increasing by up to 210%.

App A	MiW
moses	547.00%
masstree	142.83%
img-dnn	10.20%
specjbb	9.00%
xapian	8.51%
silo	8.39%

Table 3.3: The degrees of MiW (more-is-worse) over pairs of applications (App A/B) which divide up LLC. The MiW degree is measured by comparing the tail (95th percentile) latency of App A when it occupies the maximum share of LLC (numerator) with the one when it performs best over all possible LLC shares (denominator) through CP. For App B, 403.gcc has been used for all cases.

3.4 Workload Analysis

To analyze the characteristics of the workload, the top-down analysis described in Section 2.3.2 was used [63]. First, among the SPEC CPU2006 benchmarks, applications sensitive to cache capacity were selected through previous studies [61], [62], [63], [68] and experiments. Since we used the spec rate, multi-core results were important. For example, 482.sphinx3 of SPEC

CPU2006 had a large L3 bound, and no MEM bound when operated in single-thread mode, but when operated in multi-core, the L3 bound decreased, and the MEM bound increased significantly [63]. This phenomenon can be seen because of competition among threads against shared LLC.

We focused on applications where the backend bound during top level breakdown is much larger than the frontend bound, and the memory bound is very large compared to the core bound during backend level breakdown. Among them, attention was paid to applications that are sensitive to cache capacity and memory bandwidth through memory level breakdown. Representatively, there are gcc, mcf, omnetpp, astar, xalancbmk, milc, leslie3d, soplex, GemsFDTD, lbm, wrf, and sphinx3. These applications have a very high memory access ratio compared to computation (minimum 1:1.6, maximum 1:9.8), are sensitive to memory bandwidth, and have a common feature that the memory bound rises rapidly when there is competition for LLC. Additional noteworthy applications are gcc and omnetpp, both of which have higher store bounds than other applications.

For the cache sensitivity of a single application, 20 cores were used to measure the average IPC and MPKI per core. The result showed a change in performance according to the cache capacity similar to previous studies, and 144 application combinations were created with these 12 applications and used in the experiment. Only the results of application combinations in which MiW was most

prominent were shown, and in most of those combinations, MiW of around 2% occurred.

The applications used in SPEC CPU2017 were also selected based on the same criteria as the applications in SPEC CPU2006 [66], [67].

3.5 The Root Cause of the MiW Phenomenon

To understand the root cause of MiW, we first pay attention to the fact that MiW occurs when applications stress the main memory bandwidth of a system. Figure 3.7 shows the relationship between the bandwidth load and the observed latency of a main memory system with the peak bandwidth of 76.8GB/s specified in Table 3.1.

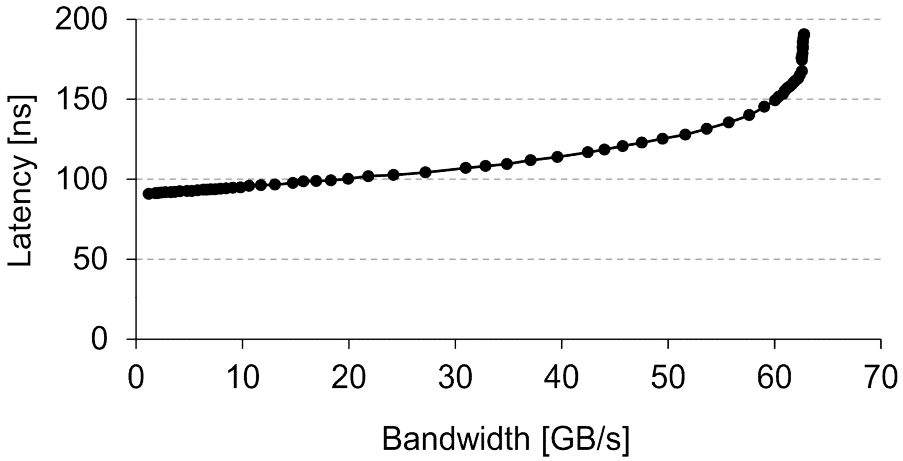


Figure 3.7: Load-latency values of the tested system (Table 3.1) with 76.8GB/s of max main memory bandwidth. Latency rises rapidly when system bandwidth gets closer to the peak.

Main memory access latency values increase slowly when the memory system is lightly loaded, but they increase rapidly as the load gets closer to the theoretical peak bandwidth, similar to interconnection networks [9]. When a larger portion of LLC capacity is allocated to the synthetic workload group-A in Figure 3.4, the other workload group-B receives smaller LLC capacity, experiences higher LLC MPKI, stresses main memory bandwidth that is shared between group-A and group-B, and hence increases memory access latency for both group-A and group-B. In other words, when group-B stresses the main memory bandwidth due to fewer LLC ways allocated, group-A also experiences high memory access latency breaking the performance isolation between the workload groups, which is the very intention of CP.

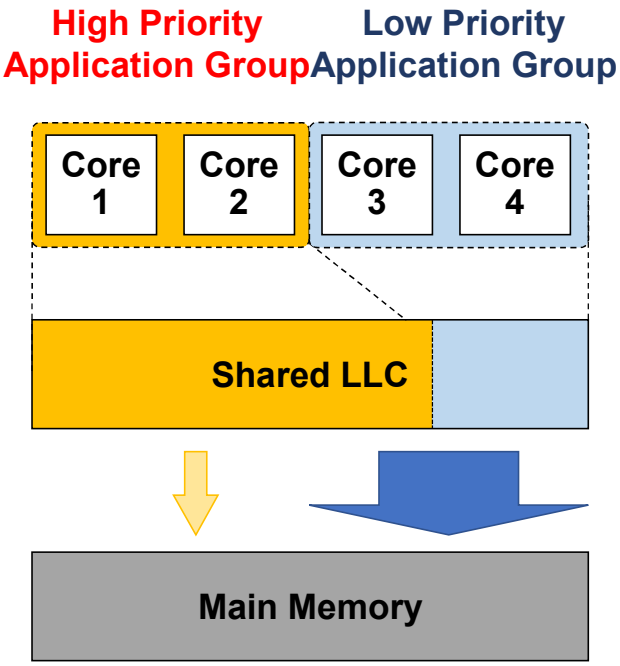


Figure 3.8: When a larger portion of LLC capacity is allocated to high priority workload group, the other low priority workload group receives smaller LLC capacity. The low priority workload group experiences higher LLC MPKI, stresses main memory bandwidth that is shared between both workload groups, and hence increases memory access latency of the entire system.

Therefore, the group with more LLC capacity (group-A) has higher memory access time for memory requests that miss LLC; this overhead can even outweigh the benefits of lower LLC MPKI due to larger LLC capacity, resulting at performance drop especially if group-A is highly sensitive to main memory latency (Figure 3.8).

It might appear as if memory requests from different applications are heading to the same destination (a memory channel) and hence these requests cannot be isolated, leading to a surge in access latency values on all the requests; but in reality, they are likely headed to different destinations. When the requests from both processes access the same target in main memory (e.g., the same DRAM bank), they all should experience high loaded access latency due to the elevated degree of queuing delay. However, different processes mostly access different targets (e.g., different DRAM banks) as modern CMPs typically have dozens of DRAM banks per channel; so, the chances that two requests from different processes access the same bank are meager[③].

Then, the reason why a surge in LLC MPKI of one process (group-B) negatively affects the performance of the other (group-A) could be due to blocking of the data path that a request handling an LLC miss experiences, a well-known problem in designing the flow control of interconnection networks when requests from different source-destination pairs share the same intermediate data path (e.g., buffers) [9]. This blocking occurs when the oldest packet in an intermediate shared buffer cannot be transferred because the next node on the route for its destination is congested, the "younger" packets in the shared buffer are blocked, resulting in a performance drop. A solution for this blocking is to virtualize

③ Techniques to partition main memory such that a bank is dedicated to a process (e.g., PALLOC [52] and [28]), can be used to ensure banks are not shared between processes.

the data path, such as virtual channels [9].

Moreover, requests from one process (group-B) can occupy a significant portion or even all of the shared intermediate data path (memory request buffer), which is a valuable/scarce resource. This limits the memory controller' s visibility of the processes (group-A and -B) with different access behaviors, and lead to a poor scheduling decision. Virtualizing the data path can help to solve this problem.

We first show that existing hardware does not have virtualized data path in memory controllers. We control main memory bandwidth demands from two groups of processes (group-C and group-D) such that group-C alone spends half the peak bandwidth of a system, and group-D alone spends the entire bandwidth. When we run group-C and group-D together, we observed that group-C burns 1/3 of main memory bandwidth, whereas group-D uses the other 2/3. If the memory requests from group-C and group-D are through virtualized data path, as group-C and group-D both have the same priority level, they should both utilize 1/2 of main memory bandwidth.

Figure 3.9 show the main-memory bandwidth utilization as the number of cores dedicated to group-C changes while group-D runs on 20 cores. The line graph indicates the theoretical portion group-C should occupy within the total memory bandwidth, calculated by the ratio of the number of cores allocated to group-C (numerator) and the sum of cores allocated to both group (denominator).

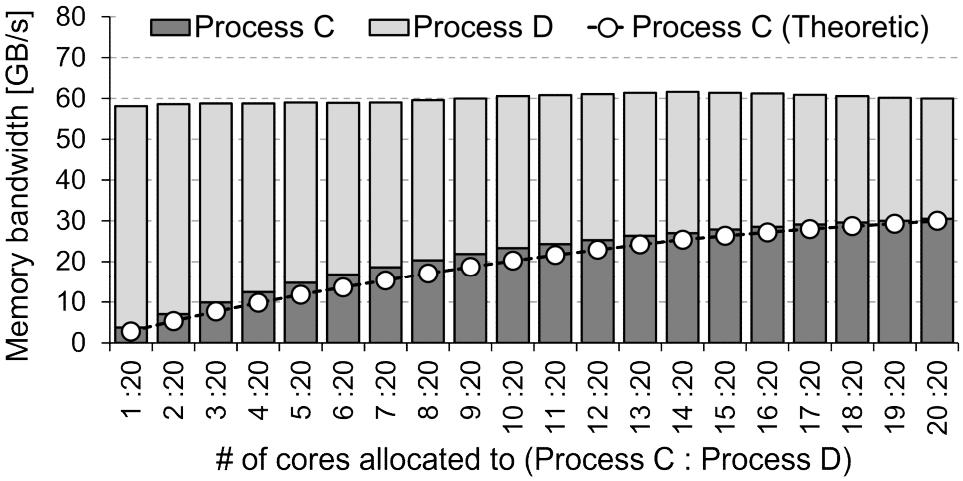


Figure 3.9: Main memory bandwidth utilization with two processes (C and D) evaluated with a 76.8GB/s peak main memory system, as the number of cores dedicated to C changes, and D alone uses the whole bandwidth of the system. The line graph indicates the ratio of the number of cores allocated to process C, over the total number of cores in use based on the total utilized memory bandwidth.

The result shows that measured bandwidth ratios match close to the theoretical values rather than group-C and group-D sharing the bandwidth equally, through which we can verify that the evaluated hardware does not have virtualized data path in the memory controllers.

3.6 Limitations of Existing Solutions

Before exploring the idea of virtualizing the data path of memory controllers, we first assess if the ideas that are already implemented in hardware (main memory bandwidth throttling [17]) or have been extensively studied before (memory scheduling considering fairness [27], [36]) can address MiW. Through experiments with the latest HW and simulation, we observe these existing solutions cannot eliminate MiW.

3.6.1 Memory Bandwidth Throttling

The latest Skylake-based [11] Xeon systems support a feature named Memory Bandwidth Allocation (MBA) [17], which limits the memory bandwidth dedicated to each group (class). We evaluated a system with a single socket Skylake server with 24 physical cores (Hyper-Threading enabled), 33MB of shared LLC with 11 ways, and 21.3GB/s of peak main memory bandwidth. MBA limits memory bandwidth by the granularity of 10% (we used the linear mode [17]).

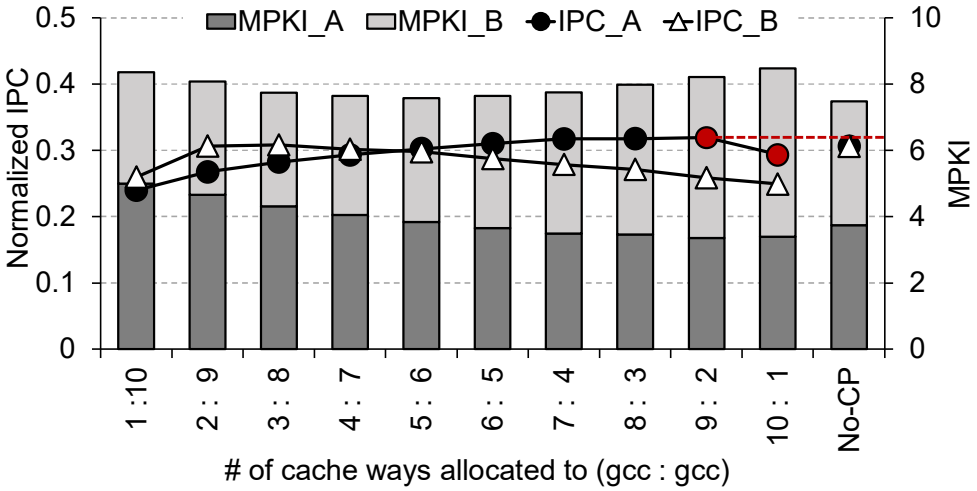


Figure 3.10: The impact of Memory Bandwidth Allocation (MBA) on 403.gcc-403.gcc. Even if MBA was used, MiW phenomenon occurred when more than 9 LLC ways assigned to group-A, showing MiW phenomenon could not be completely solved.

Figure 3.10 shows the normalized IPC and stacked-up MPKI values of a pair of 403.gcc and 403.gcc, similar to the experiments in Chapter 3.1 except that MBA is enabled here. We allocated 90% of bandwidth allocation (the higher, the more bandwidth allocated) to group-A and 10% to group-B. The result shows MiW is still observed for the machine with MBA. We evaluated different bandwidth allocation ratios (e.g., 10%/90% and 50%/50% to group-A/-B), but MiW still persists.

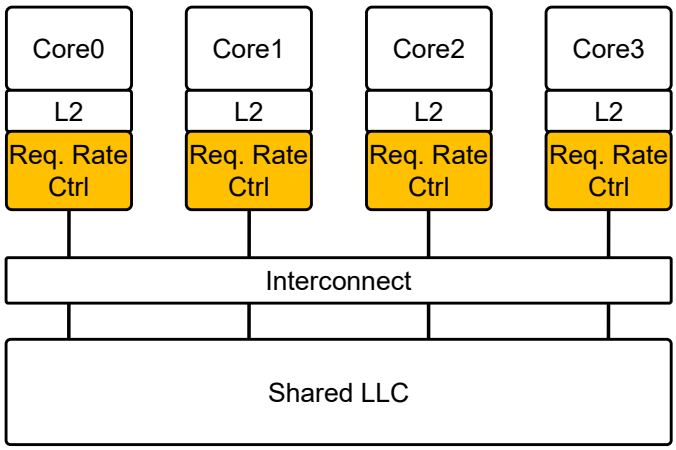


Figure 3.11: MBA controls memory bandwidth indirectly and approximately. MBA places a programmable rate controller in L2 MSHR, a boundary between private L2 caches and shared LLC

However, if we change the configurations such that main memory is not bandwidth saturated by either decreasing SPEC rate, increasing peak main memory bandwidth by populating more channels (Skylake supports up to 6 channels per socket), or lowering the bandwidth allocation values of MBA to all the application groups, MiW mostly disappears. This also indicates that the blocking in congested memory controllers is a likely source of MiW.

The memory bandwidth throttling looks like a plausible solution, but MBA has a limitation in that it controls memory bandwidth indirectly and approximately [17]. MBA places a programmable rate controller in L2 MSHR (Miss Status Holding Resister), a boundary between private L2 caches and shared LLC (Figure 3.11). This

enables per-core rate control (source throttling) without introducing virtualized data path. However, as L2 misses are then filtered through LLC (whose miss rates are hard to predict as it is shared among many cores), this indirect bandwidth control is inevitably approximate. Therefore, MBA must conservatively limit memory bandwidth to prevent the blocking (over-throttling), and hence the performance of all the application groups would be sub-optimal due to this main memory bandwidth underutilization.

3.6.2 Fairness-aware Memory Scheduling

Among the proposals of providing fairness on top of memory access scheduling (the control part of a memory controller), we selected two representative ones and tested if they can address MiW.

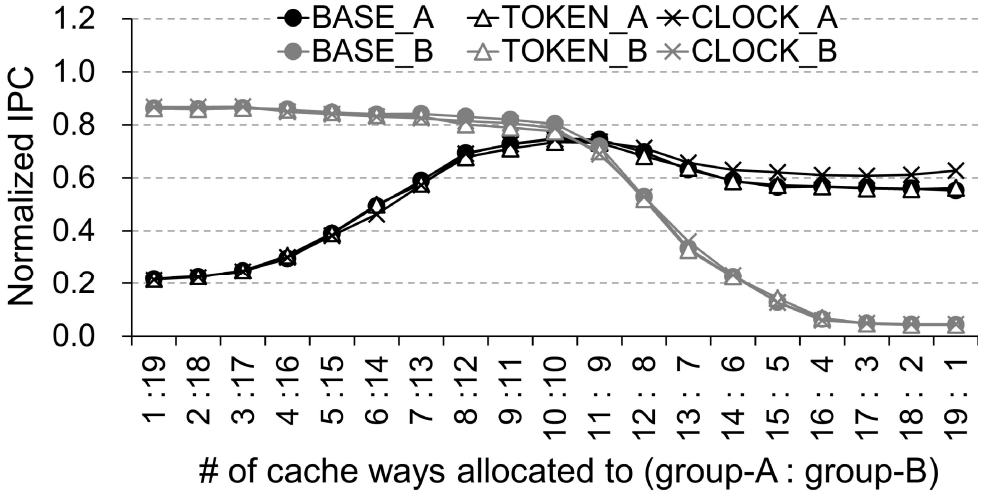
First, we chose the token bucket algorithm (TOKEN), which was originally introduced as an arbitration method for interconnection networks [27], [38], [54]. For TOKEN, each request can be processed when it has a matching token in the respective bucket (each for the corresponding group). A token generator distributes tokens to different buckets at the rates proportional to the fractions allocated to different groups.

Second, a request prioritization method, which gives priority based on a virtual clock (CLOCK) [36], is a memory version of deadline-based arbitration frequently adopted in interconnection

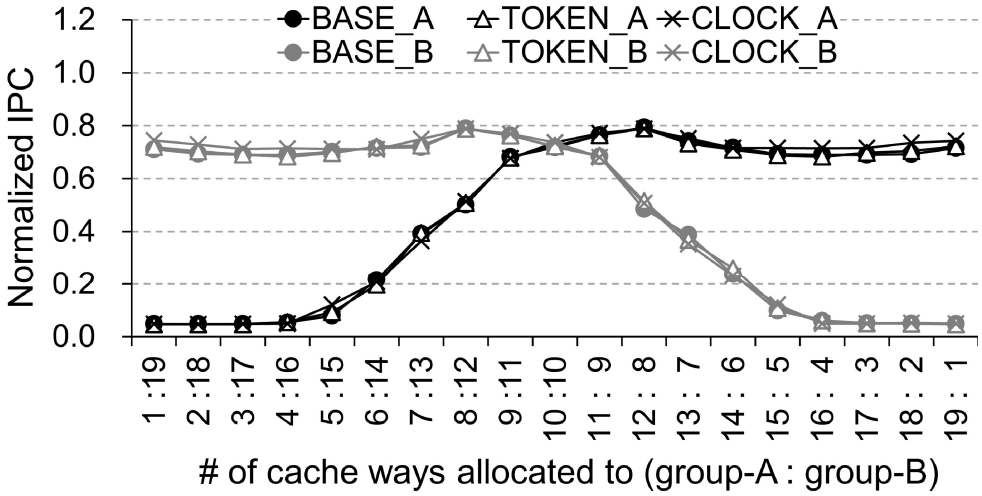
networks. CLOCK prioritizes 1) ready commands, 2) column-level commands, and 3) commands with the earliest virtual finish time. The virtual finish-times of the DRAM commands from each memory request are calculated based on prior work [36]. To prevent priority inversion by bank priority chaining, after a DRAM bank has been restored in the course of row activation (around 32ns in modern DRAM devices), rule 3) is applied first overrule rule 2) among the requests heading to the same bank. We set both TOKEN and CLOCK to treat all the application groups equally.

Because these schemes are not implemented in existing hardware, we used simulation, whose setup is detailed in Chapter 4.3.1. Two benchmark pairs from SPEC CPU2006 [45] are used (Figure 3.12). Both TOKEN and CLOCK perform on par with or better than the baseline memory-access scheduling scheme of FR-FCFS (BASE in Figure 3.12), but MiW persists.

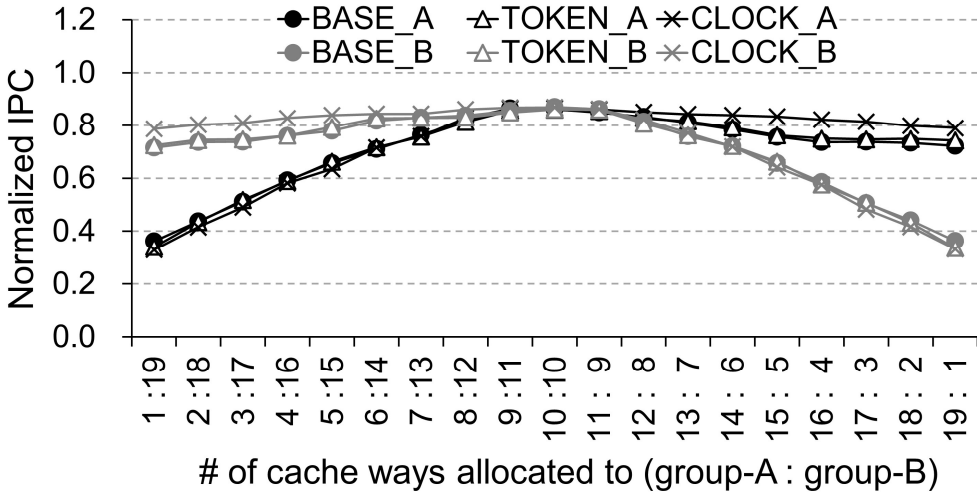
When two application groups are executed, TOKEN keeps each group from using more than half of the system' s peak memory bandwidth. Therefore, TOKEN restricts a group' s memory bandwidth only when it requires more than half of the system' s peak memory bandwidth, allowing both groups to utilize memory bandwidth more fairly. CLOCK prioritizes a request with the earliest deadline (finish-time) and hence tries to divide the system' s memory bandwidth equally for each group. However, because neither TOKEN nor CLOCK eliminates the blocking problem when the main memory system is bandwidth saturated, MiW does not disappear.



(a) 403.gcc – 473.astar



(b) 403.gcc – 403.gcc



(c) 523.xalancbmk – 523.xalancbmk

Figure 3.12: Simulation results of augmenting the default memory access scheduling (FR-FCFS [40], BASE) with token-bucket (TOKEN [27]) and virtual clock (CLOCK [36]) algorithms. The simulation results show these fairness-aware memory scheduling algorithms cannot resolve MiW.

These and other recent memory access scheduling proposals [4], [47], [48] pursue fairness in scheduling by limiting the number of consecutive requests to a specific DRAM bank, by limiting the number of reordering a request can experience to serve other requests with a higher priority, and by rotating the priority among the requests originating from different sources (e.g., cores). However, these proposals prioritize requests within a buffer; if a certain request cannot enter the memory request buffer (as the buffer are full due to blocking, for example), the scheduler cannot address the problem, and the system suffers from MiW.

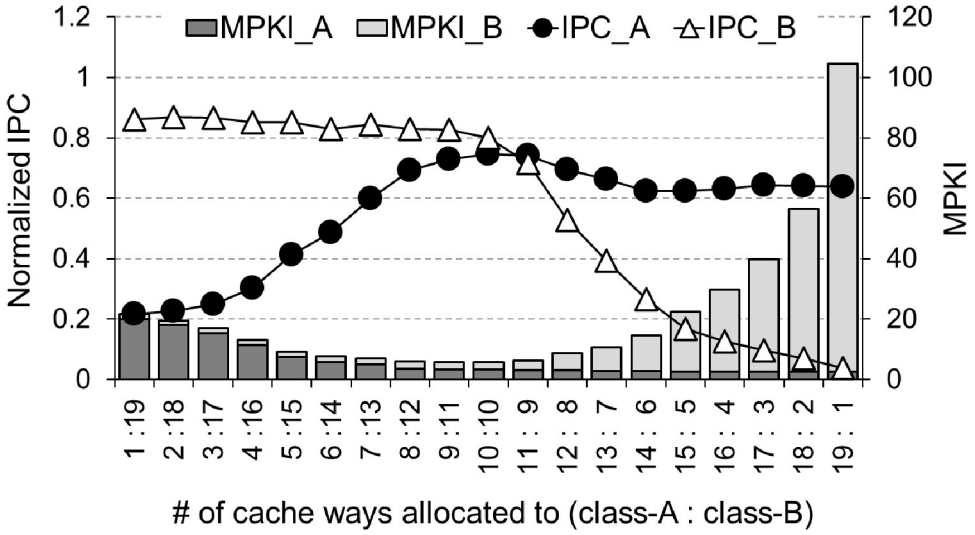


Figure 3.13: Simulation results of ATLAS [22]. The simulation result shows QoS/fairness-aware memory scheduling algorithm cannot resolve MiW.

More recent related work includes ATLAS [22]. ATLAS prevents memory-intensive processes from monopolizing memory bandwidth by prioritizing requests from the least acquired memory service thread. Although effective for QoS, ATLAS was originally designed to maximize aggregate throughput. ATLAS performs equal or better than the baseline, but MiW was not eliminated.(Figure 3.13)

Chapter 4

Virtualizing Memory Channels

In Chapter 3, we observed MiW, which is a phenomenon leading to performance degradation when more LLC capacity is allocated. We have uncovered the root cause of MiW and found that MBA or other proposals do not solve the MiW completely. To alleviate MiW, we propose to virtualize the data path of memory controllers.

We explore possible design spaces for mVC and propose four possible buffer allocation strategies in Chapter 4.2 and evaluate the impacts of each strategy in Chapter 4.3. Also, through case studies, we show the possibility to reduce the overall system cost using mVCs with a proper memory request queue size and LLC capacity while satisfying the target performance of latency-critical workloads even when executed with multiple workloads together.

4.1 Memory Virtual Channel (mVC)

To prevent/alleviate the blocking in main memory systems, we propose to virtualize the data path of memory controllers (MCs) by providing a separate request buffer per group (class) of LLC. As opposed to the previous works utilizing per-source (CPU vs. GPU) [4] or per-thread [36] request buffers, we use per-group (per-class) separate buffering called memory virtual channel (mVC).

Per-source separate buffering is too coarse-grain as it does not separate requests from different cores within CPU or GPU, and per-thread separation is too expensive as the number of hardware threads in modern shared-memory chip multiprocessors can exceed a few hundred. We assume that NoC (network-on-chip) is not a source of blocking[④]; if so, it should be virtualized as well or support other blocking prevention feature (e.g., bufferless flow control [34]).

Similar to Intel MBA, we align the class of MC and that of LLC; therefore, a group (class) of applications have both dedicated LLC capacity and MC's buffer (queue) space. As opposed to the source throttling of Intel MBA, which cannot prevent blocking in MCs because it does not precisely know the amount of traffic filtering by LLC, mVC guarantees blocking prevention. All data path within a

---

④ To the best of our knowledge, the NoC prior to Intel Skylake-based Xeon systems implements a ring NoC with prioritized arbitration and thus, blocking does not occur within the NoC itself.

MC must be virtualized. If a MC has multiple stages of buffers (e.g., staged memory scheduling [4]), they all should be virtualized (separated by groups). Otherwise, this un-virtualized portion of data path becomes the source of blocking.

The control part of a MC (i.e., memory access schedulers) must be augmented to provide fairness among the groups/classes (see Figure 4.2.(a)). For example, FR-FCFS [40] gives a higher priority to a ready request (which can be serviced with a RD or WR DRAM command without any timing constraint) over non-ready requests, on top of the baseline priority rule of first-come-first-serve (FCFS). With the mVC support, there should be a round-robin arbitration logic between the classes, which should be the highest priority tier compared to both FR and FCFS.

A MC with mVCs requires a round-robin arbitration logic, which we refer to as mVC arbiter, which selects a DRAM command at a given cycle among the command candidates from different groups (classes). This round-robin arbiter responds with a single grant. Any buffer without an available DRAM command is simply skipped over and ignored by the mVC arbiter. However, as opposed to NoC arbiters, an arbiter grant is not released after servicing a single DRAM command but held until a column-level (RD/WR) command is served. This ensures that if two (or more) request buffers target the same DRAM banks, it avoids DRAM row-buffer conflicts by continuously issuing a sequence of ACT and PRE commands, avoiding deadlocks, and providing fairness.

The multiple per-group request buffers do not necessarily increase the cost (in terms of storage) as the total amount of storage for the buffers are held constant; the only difference is the amount of storage per request buffer which can be smaller compared to the baseline request buffer. The additional cost for the mVC arbiter is also relatively small because the number of groups is usually much smaller than the aggregate number of entries in the request buffer.

4.2 mVC Buffer Allocation Strategies

One design question for mVC is how to allocate buffer space in the memory request buffers to the different mVCs. Figure 4.1 and Figure 4.2 compares the conventional memory request buffering (Figure 4.1) with the following four different buffer allocation strategies for mVC (Figure 4.2.(a), (b), (c), (d)):

- Static (mVC-STATIC): A simple strategy is to partition the request buffer statically in the same size among all the mVCs. While preventing starvation of either flow, this scheme may lead to underutilization of request buffers when the memory request rate from the LLC is highly skewed between the two groups (Figure 4.2.(a)).

- Proportional (mVC-PROP): A second strategy is to allocate buffers to each group in proportion to its share of LLC ways. For example, if group-A and group-B are allocated 15 and 5 LLC ways, they receive 9 and 3 entries in the request buffer, respectively (Figure 4.2.(b)). The rationale of this strategy is to partition storage resources along the shared memory access path by the same ratio. It can alleviate MiW by preventing the group with fewer resources (say, group-B) from flooding the entire request buffer and slowing down the other group.
- Inverse Proportional (mVC-INVPROP): The next strategy is to allocate buffers to each group inversely proportionally to its share of LLC ways. In contrast to mVC-PROP, group-A and group-B receive 3 and 9 entries in the request buffer when 15 and 5 LLC ways are allocated to them, respectively (Figure 4.2.(c)). Because groups that receive fewer LLC ways are likely to incur more LLC misses, this strategy tends to allocate more buffers to groups incurring LLC misses more frequently.
- Dynamic (mVC-DAMQ): We also consider a dynamic buffer allocation strategy based on DAMQ (dynamically allocated multi-queue) [9]. DAMQ partitions the request buffers dynamically among mVCs based on the request rate of each mVC. By partitioning the request buffer into shared and dedicated regions, a deadlock which would occur when a

memory-intensive workload claims all the buffer entries can be avoided. The shared region is dynamically allocated based on demands; the dedicated region is equally partitioned and dedicated to each mVC. A mVC first uses its dedicated region to store memory requests. Once its dedicated region is full, it claims an entry from the shared region for the next memory request (Figure 4.2.(d)).

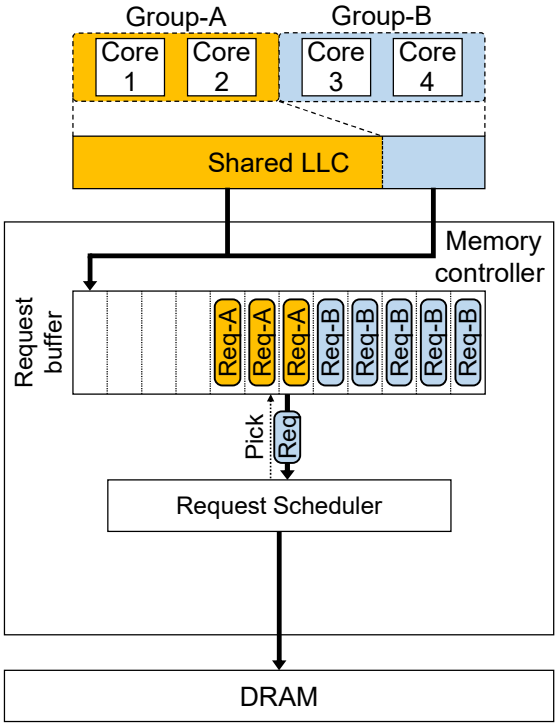
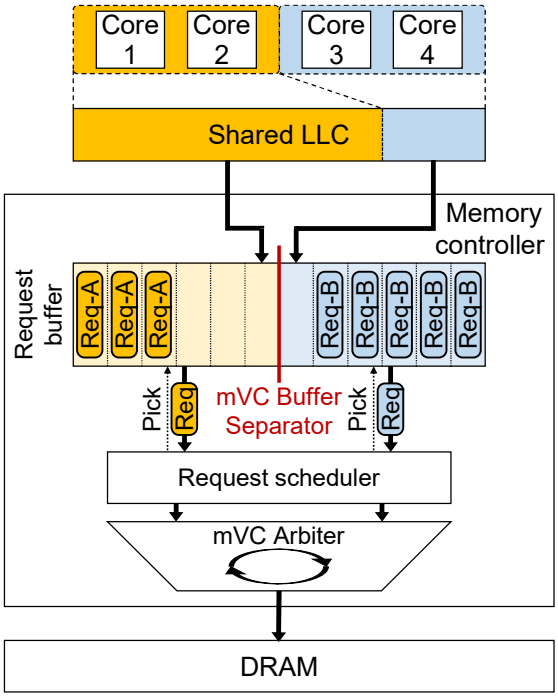
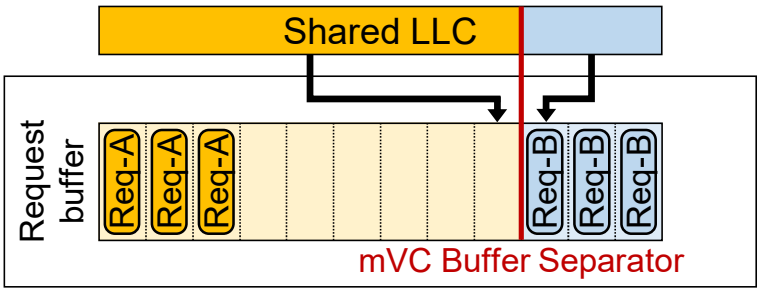


Figure 4.1: Conventional cache-oblivious memory request buffering.

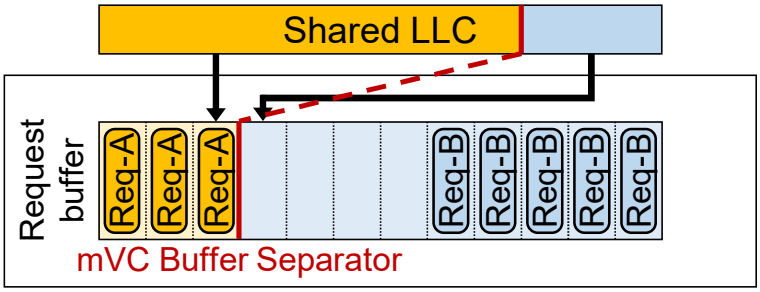


(a) Static (mVC-STATIC)

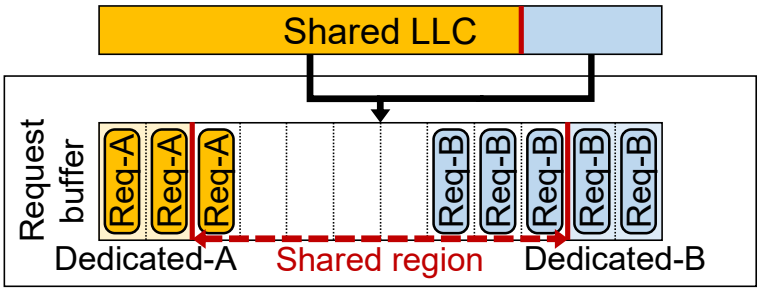
Figure 4.2: Four buffer allocation strategies for mVC; (a) Static.



(b) Proportional (mVC−PROP)



(c) Inverse Proportional (mVC−INVPROP)



(d) Dynamic (mVC−DAMQ)

Figure 4.2: Four buffer allocation strategies for mVC; (b) proportional, (c) inverse proportional, and (d) dynamic.

4.3 Evaluation

To evaluate the impact of mVCs, we model a CMP system having CP, virtualized memory channels, and MBA. We first reproduced MiW through simulations and evaluate the effectiveness of virtualizing memory channels.

4.3.1 Experimental Setup

We simulated a CMP system to evaluate the effectiveness of mVCs, whose parameters are summarized in Table 4.1. McSimA+ [2] simulator was modified for the simulation. The baseline memory controller has a 20-entry request buffer and adopts FR-FCFS [40] as a memory request scheduling policy and adaptive open policy (which is also adopted at Intel Xeon processors) as a DRAM page management policy.

SPEC CPU2006 [45] and SPEC CPU2017 [46] benchmark suites were used for evaluation. Simpoint [43] was used to extract the most representative simulation points of each application, each including 100 million instructions. We sorted and selected cache-sensitive applications in SPEC CPU2006 and SPEC CPU2017 benchmarks and used them for evaluations.

Parameter	Value
# of cores	16 cores
# of MCs	1 MC
Coherence policy	MOESI
Cache line size	64B
Per Core:	
Frequency	3.6GHz
Issue/commit width	4/4 slots
Issue policy	Out-of-Order
L1 I/D $ type/size/associativity	Private/32KB/8
L2 $ type/size/associativity	Private/256KB/8
L3 $ type/size/associativity	Shared/40MB/20
Per DDR4-2400 memory controller:	
# of channels	1
Request queue size	20 entries
# of ranks per channel	2
Bandwidth per channel	19.2GB/s
Scheduling policy	FR-FCFS [40]
DRAM page policy	Adaptive Open [16]

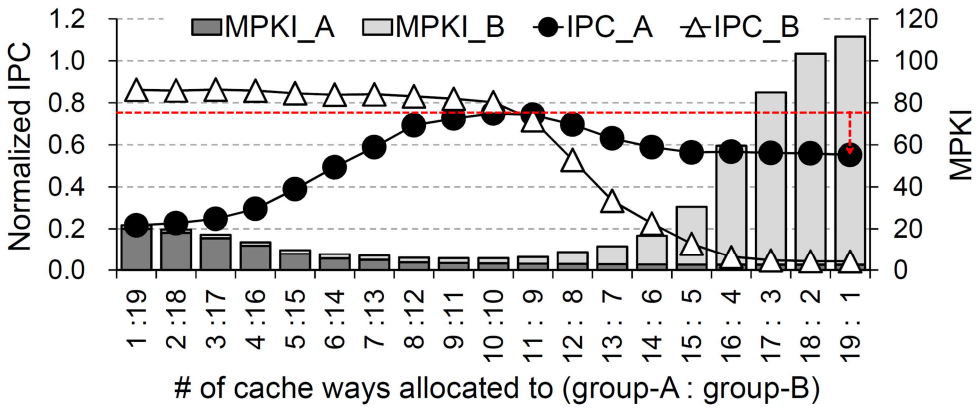
Table 4.1: Parameters used in the simulated system.

We compare four buffer allocation strategies for mVC: mVC-STATIC, mVC-PROP, mVC-INVPROP, and mVC-DAMQ. For static buffer allocation (mVC-STATIC), 10 entries are allocated to each mVC with two mVCs, which is equal to a total memory request

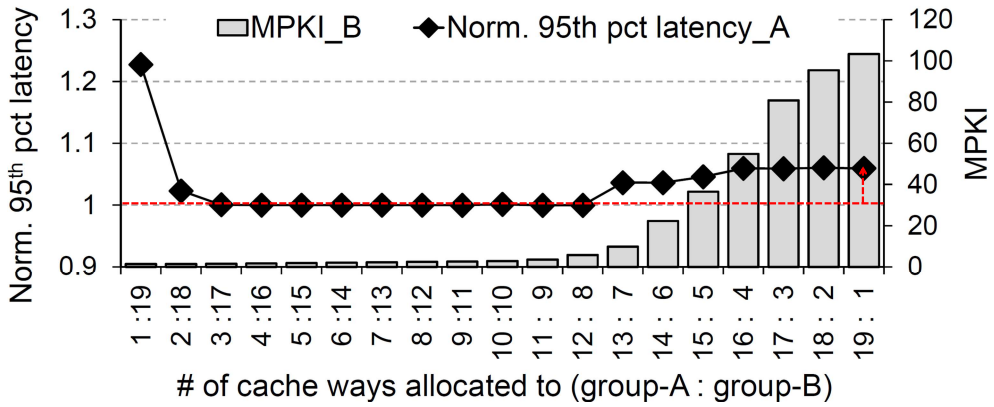
buffer size of 20. For proportional buffer allocation (mVC-PROP), the number of buffer entries allocated to each mVC is based on the number of LLC ways allocated to each mVC. On the contrary, for inverse proportional buffer allocation (mVC-INVPROP), the number of buffer entries allocated to each mVC is (20 - the number of LLC ways allocated to each mVC). We also evaluated mVC with dynamic buffer allocation (mVC-DAMQ) based on 80% shared region size in the request buffer.

4.3.2 Reproducing Hardware Results

Before evaluating the proposed mVCs, we reproduced the hardware results through simulation (Figure 4.3). Xapian in Figure 4.3.(b) is an application in TailBench, and the group-A consists of its single-threaded instance. Both normalized IPC and 95th percentile latency is normalized based on those when each benchmark runs alone with 20 LLC ways allocated. Similar trends as the hardware results are observed (Figure 3.1, Figure 3.2, and Table 3.3) and clearly show MiW. The other case also matches with Table 3.2.



(a) 473.astar for group−A and 403.gcc for group−B



(b) xapian for group−A and 403.gcc for group−B

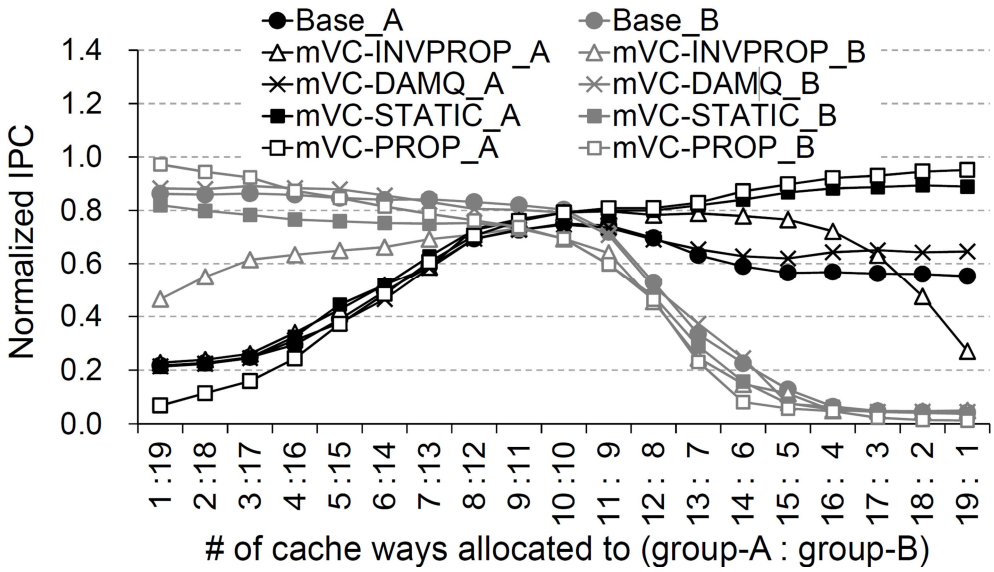
Figure 4.3: Simulation results on SPEC CPU2006 and TailBench showing trends similar to hardware experiments, reproducing MiW.

4.3.3 Mitigating MiW through mVC

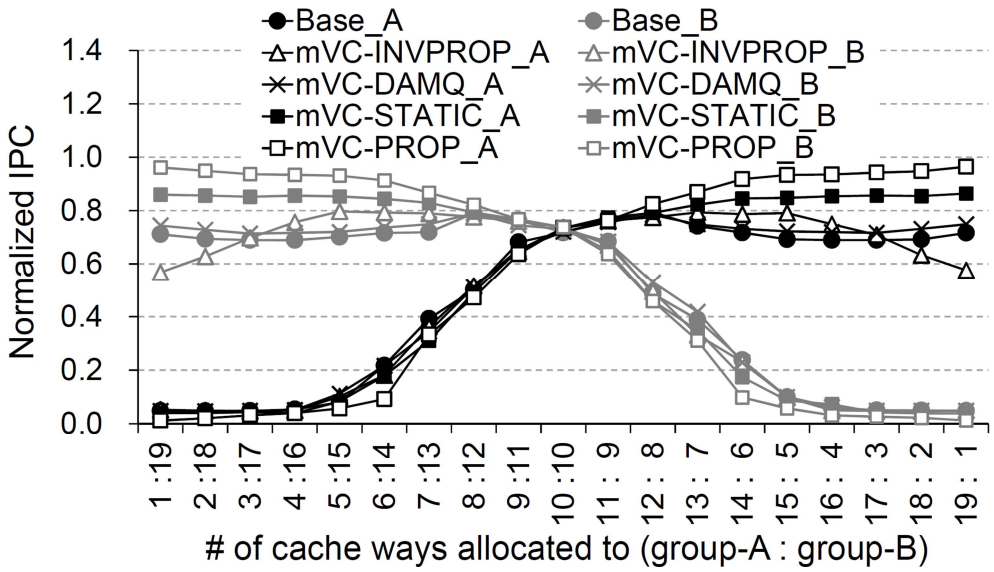
We evaluate the effectiveness of virtualizing the data path of memory channels by executing multi−programmed workloads on

the simulator. Figure 4.4 shows the IPCs of three workload pairs that demonstrate MiW in Chapter 3.1 (Table 3.2 and Table 3.3), normalized to the IPCs with standalone execution. We compare four buffer allocation strategies for mVC discussed in Chapter 4.2: static (mVC-STATIC), proportional (mVC-PROP), inverse proportional (mVC-INVPROP), and dynamic (mVC-DAMQ). Because there are 16 cores, we executed each benchmark with a rate of 8.

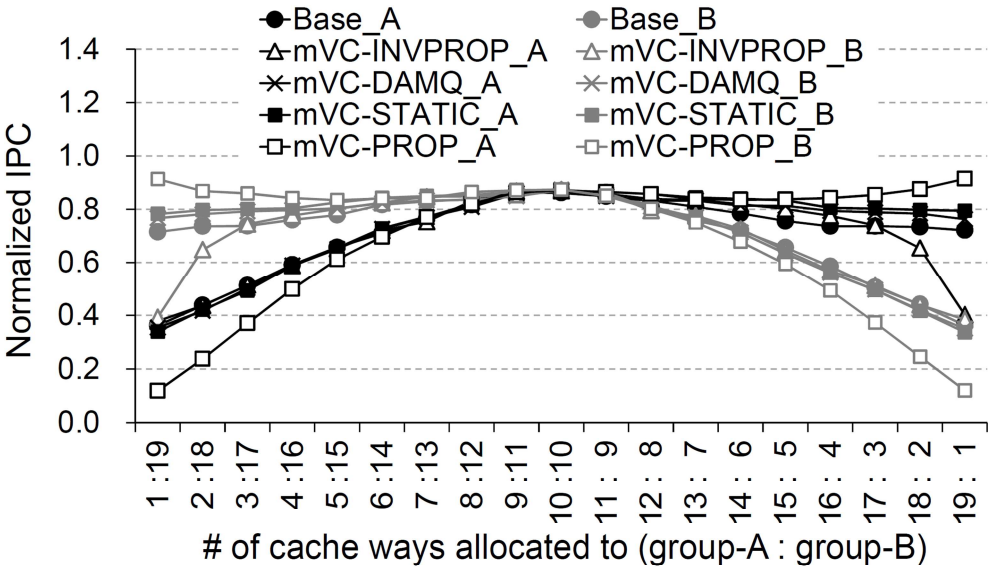
We made the following key observations. First, mVC effectively addresses the blocking problem except for mVC-INVPROP and mVC-DAMQ. As group-A gets allocated with more LLC ways in the baseline without mVC, the requests from group-B flood the request buffer to cause starvation of group-A. With mVCs, however, group-A has a guaranteed share of the request buffer entries and a fair chance for DRAM accesses via round-robin scheduling, alleviating the problem of blocking and eliminating MiW. For example, Figure 4.4.(a) shows the results using a 473.astar-403.gcc pair. With mVC-PROP and mVC-STATIC, 473.astar achieves 95.2% and 86.4% of the IPC of standalone execution, respectively, while the baseline achieves only 75.0% without mVC due to MiW. MiW is also eliminated in Figure 4.4.(b) and (c). By recovering lost IPC from MiW, this opens additional opportunities for consolidating workloads requiring an IPC service-level objective (SLO) [30] with other best-effort workloads.



(a) 473.astar for group-A and 403.gcc for group-B



(b) 403.gcc for group-A and 403.gcc for group-B



(c) 523.xalancbmk for group-A and 523.xalancbmk for group-B

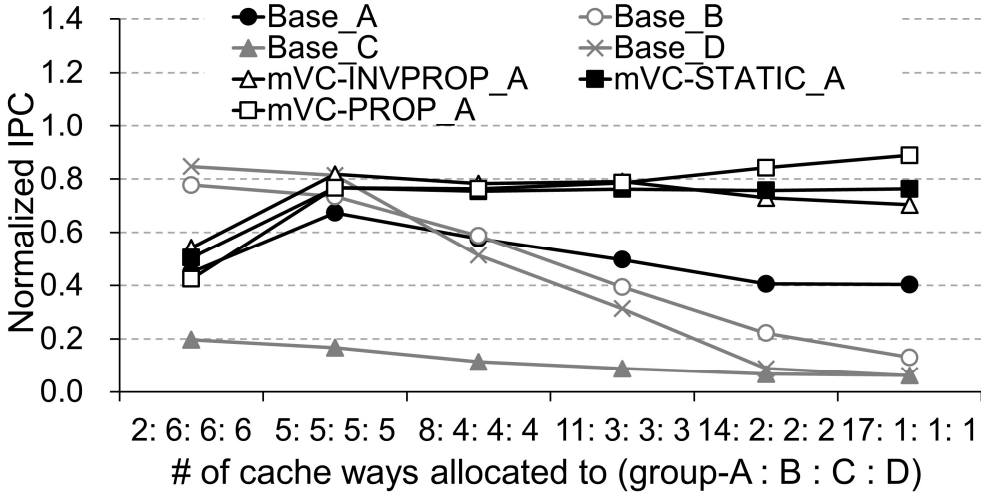
Figure 4.4: Simulation results of mVC with different memory request buffer allocation policies: mVC-STATIC, mVC-PROP, mVC-INVPROP, and mVC-DAMQ. The normalized IPC is normalized based on those when each benchmark runs alone with 20 LLC ways allocated.

Second, mVC-PROP more effectively eliminates MiW than mVC-STATIC at the cost of penalizing the group with fewer resources, while mVC-DAMQ and mVC-INVPROP fail to eliminate MiW. In mVC-PROP, as group-A receives more LLC ways, more request buffer entries are allocated to it, yielding higher memory throughput due to a larger memory scheduling window. mVC-STATIC allocates memory requests fairly, which may increase

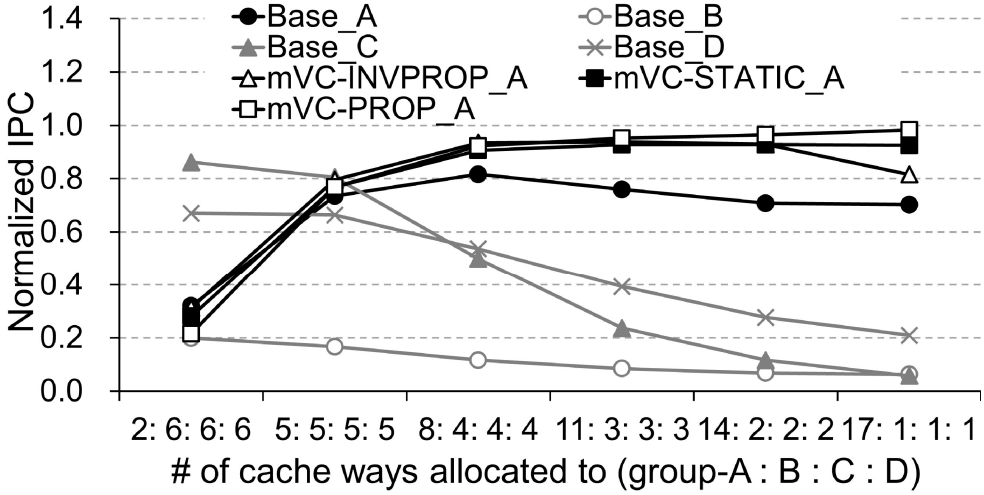
system-wide throughput in some cases. Assuming an 80:20 division of the shared and dedicated regions, mVC-DAMQ performs slightly better than the baseline, but cannot eliminate MiW because group-B experiences a high LLC MPKI to flood the shared region of the request buffer, leading to starvation of group-A. If the dedicated region is expanded to alleviate this problem, mVC-DAMQ eventually behaves like static buffer allocation (mVC-STATIC) to lose the benefits of dynamic allocation. mVC-INVPROP allocates buffer entries in an opposite way of mVC-PROP. Therefore, in contrast to mVC-PROP, which eliminates MiW, mVC-INVPROP can deteriorate MiW by allocating fewer buffer entries to the group. This trend is clearly observed in our simulated cases.

4.3.4 Evaluation on Four Groups

So far, we have only considered the case of two workload groups. This time, we investigate the case of expanding to four workload groups instead of two. Since there were only a few cache ways, more than four workload groups were hard to evaluate. We evaluated for the cases where the 20 cache ways were assigned as 2:6:6:6, 5:5:5:5, 8:4:4:4, 11:3:3:3, 14:2:2:2, and 17:1:1:1 to group-A, B, C, and D respectively. We evaluate the effectiveness of mVC, virtualizing the data path of each group by executing multi-programmed workloads on the simulator. The simulation results are shown in Figure 4.5.(a) and (b).



(a) xalancbmk, astar, omnetpp, gcc were used for group−A, B, C, and D respectively.



(b) astar, omnetpp, gcc, xalancbmk were used for group−A, B, C, and D respectively.

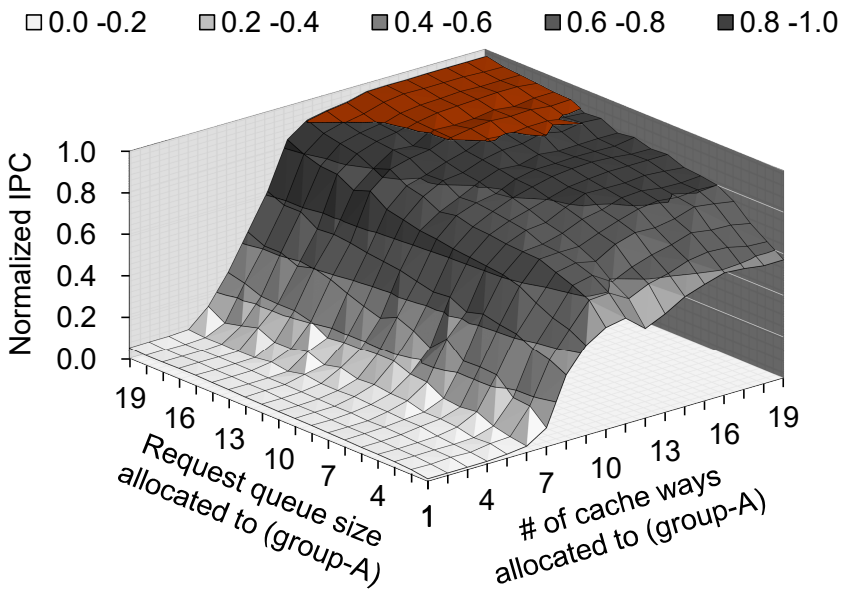
Figure 4.5: Simulation results in the case of 4 groups instead of 2 groups. Simulated for mVC with different memory request buffer allocation policies: mVC−STATIC, mVC−PROP, and mVC−INVPROP. The normalized IPC is normalized based on those when each benchmark runs alone with 20 LLC ways allocated.

For four group cases, we compare mVC-STATIC, mVC-PROP, and mVC-INVPROP. Similar observations could be made with two groups. mVC effectively addresses the blocking problem for four groups except for mVC-INVPROP, like the case of two groups. It is also not surprising that mVC-PROP mitigates MiW more effectively than mVC-STATIC.

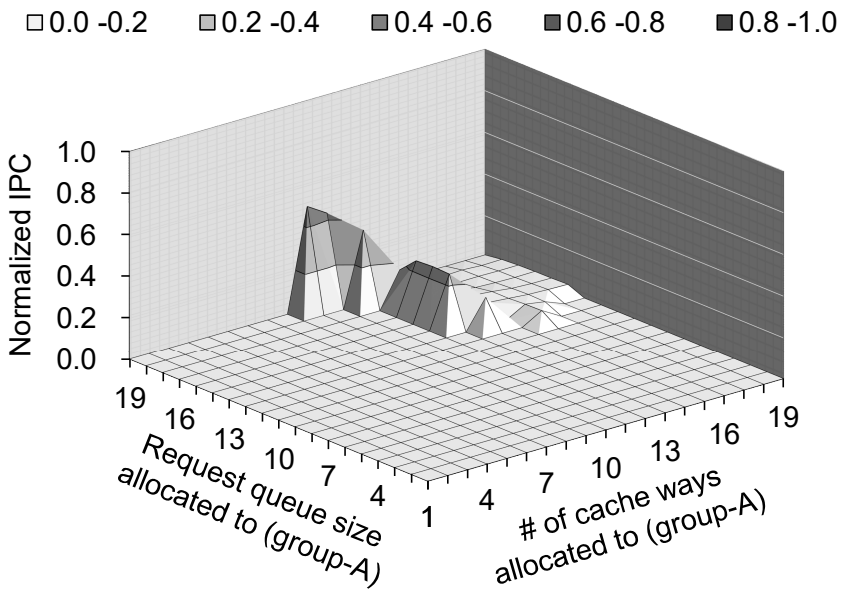
4.3.5 Potentials for Operating Cost Savings with mVC

mVC provides another knob to control resource allocation between two (or more) groups of applications. Figure 4.6 shows the results of a two-dimensional parameter sweeping for a 403.gcc-403.gcc pair, which demonstrates the greatest degree of MiW among the three SPEC CPU benchmark pairs we evaluate. X- and Y-axis represent the number of LLC ways allocated to group-A and the number of request buffer entries allocated to group-A, respectively. The Z-axis represents the corresponding IPC of group-A or B, which is normalized by its stand-alone IPC.

Figure 4.6.(a) and 4.6.(b) show the IPC normalized to standalone execution for group-A and group-B, respectively. As we run two copies of the same application, Figure 4.6.(a) and 4.6.(b) have the same shape but are oriented to the opposite direction (i.e., (x, y) = (1, 1) in (a) has the same IPC with (x, y) = (19, 19) in (b)).



(a) Simulation result of 403.gcc−403.gcc (group−A; 403.gcc)



(b) Simulation result of 403.gcc−403.gcc (group−B; 403.gcc)

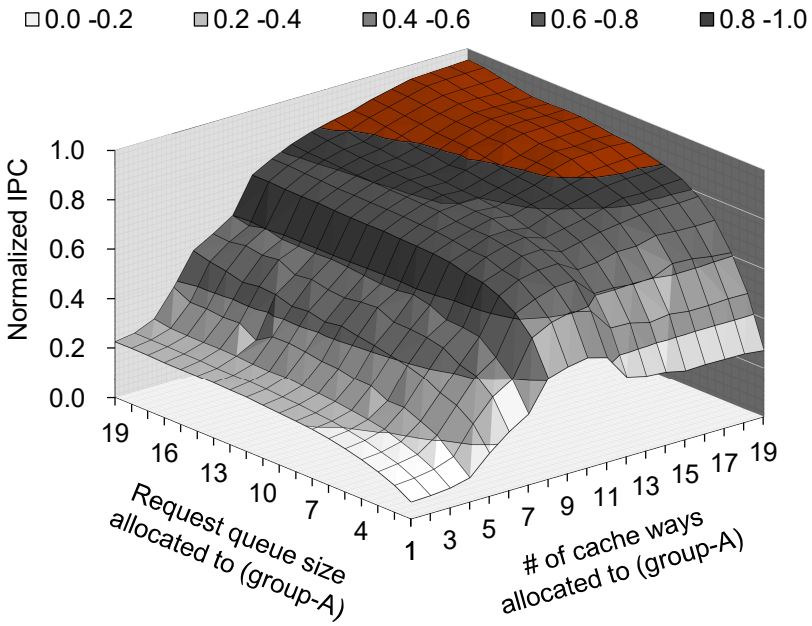
Figure 4.6: Simulation result of 403.gcc−403.gcc, showing the design space of LLC ways allocated to 403.gcc (group−A) and

memory request queue size allocated to 403.gcc (group- A). The colored region in (a) displays the design space where satisfying normalized IPC higher than 0.9, and (b) shows only the corresponding remaining region of 403.gcc (group-B).

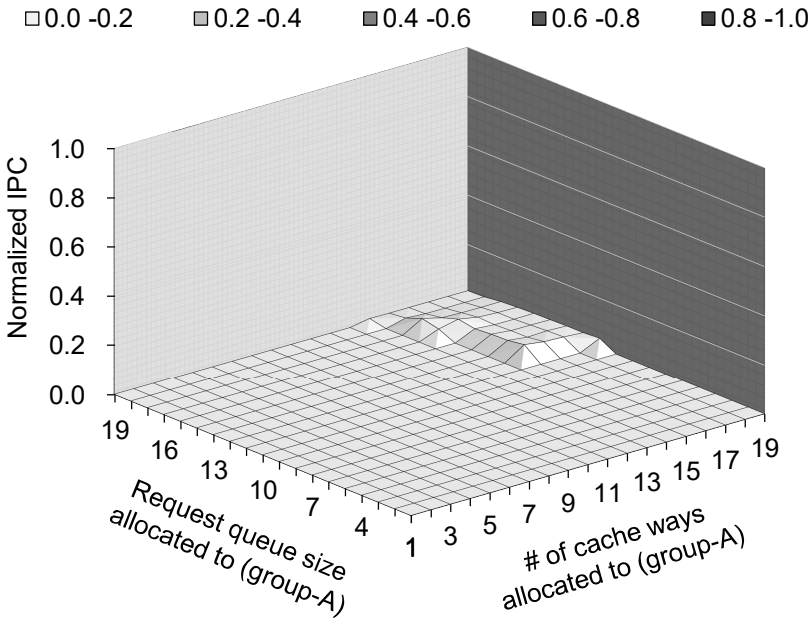
As expected, as the more LLC ways and buffer entries are allocated, the better performance is achieved. We can also observe the performance is more sensitive to the number of LLC ways allocated than the queue size. Note that the configurations that yield >90% of the standalone IPC for group-A are colored in red in Figure 4.6.(a) and that we only show the IPCs of the corresponding configurations for group-B in Figure 4.6.(b).

With a simplifying assumption of (1) cache ways being the only knob we control for resource allocation and (2) a service level objective (SLO) of 90% of the standalone IPC for group-A (group-B has no SLO), we estimate the potential for saving machine count from workload consolidation. We further assume it takes 1,000 dedicated machines for each of the two application groups in standalone mode to satisfy the application throughput requirement. From Figure 4.6, we can select an appropriate configuration of the number of LLC ways and that of buffer entries to meet the SLO target for group-A and also to maximize the throughput of group-B. For example, if we choose the point of 49.1% IPC for group-B, which is the best IPC achievable while providing a 90% IPC for

group-A, we can run group-A and group-B concurrently on 1,111 machines for group-A and group-B, and dedicate 454 extra machines to group-B to maintain the same throughput as 2,000 dedicated machines. This consolidation is only possible with mVC because, without it, group-A cannot satisfy the IPC SLO in a consolidated machine due to MiW. Thus, mVC can save 21.8% of machines compared to the baseline with LLC partitioning only, which would still require 2,000 dedicated machines to satisfy the throughput and IPC SLO. Applying the same methodology, we can save the operating cost by 7.9% and 13.3% for the other two pairs of SPEC benchmarks (473.astar-403.gcc (Figure 4.7) and 523.xalancbmk-523.xalancbmk) without violating SLO.



(a) Simulation result of 473.astar−403.gcc (group−A; 473.astar)



(b) Simulation result of 473.astar−403.gcc (group−B; 403.gcc)

Figure 4.7: Additional simulation results of the case 473.astar−403.gcc, displaying the design space where satisfying normalized IPC higher than 0.9.

Chapter 5

Related Work

5.1 Component-wise QoS/Fairness for Shared Resources

A myriad of techniques has been proposed to support quality-of-service (QoS) and fairness for shared on-chip resources, such as caches [15], [18], [31], [39], [49], [50], [51], on-chip interconnects (NoCs) [14], [25], [37] and DRAM bandwidth [22], [35], [36], [48], [52]. For caches, Suh et al. [49] introduce a dynamic monitoring scheme for the shared cache accessed by multiple concurrent threads and apply it to cache partitioning to minimize the total miss count. Qureshi and Patt [39] improve this

by using utility-based cache partitioning (UCP). CQoS [18] identifies the QoS problem in the shared LLC among concurrent threads to propose cache partitioning based on priority classifications.

Locally-fair arbitration in NoC can result in global unfairness, creating parking lot problem where remote traffic is penalized by going through more arbitrations. Recent proposals addressing this problem include Globally Synchronized Frames (GSF) [25], Preemptive Virtual Clock (PVC) [14], probabilistic arbitration [26], and LOFT [37], providing fair bandwidth allocation. Song et al. [44] observe an opposite problem in processor-interconnects of NUMA servers, where a remote flow may receive more bandwidth than highly contended local flows, calling inverse parking lot problem. However, these prior works do not address unfairness from cache partitioning.

Finally, DRAM banks and channels are other major sources of inter-thread interference. Multiple access streams from different threads may be interleaved to reduce the row buffer locality of DRAM accesses, hence degrading QoS and overall throughput. A variety of DRAM access schedulers have been proposed to recover locality and provide QoS [22], [35], [36]. For example, ATLAS [22] prevents memory-intensive processes from monopolizing the memory bandwidth by prioritizing requests from the least attained memory service thread (the expected shortest job). Though effective for QoS, ATLAS is originally designed to maximize total

throughput. MISE [48] estimates the slowdown of an application caused by memory interference through occasionally prioritizing the application over other co-running workloads; it then applies the model to devise scheduling schemes with better QoS.

However, these component-wise QoS techniques fail to provide robust performance without considering a complex interplay between different resources (e.g., LLC ways vs. DRAM bandwidth) as demonstrated in this paper and other literature [12], [30].

5.2 Holistic Approaches to QoS/Fairness

Unlike the component-wise QoS techniques, some QoS frameworks propose to manage multiple shared resources holistically. Fairness via Source Throttling (FST) [12] and GSF memory system (GSFM) [24] aim to achieve better QoS along the shared memory access path by memory injection control at each source. ASM [47] extends MISE [48] by quantifying the effect of interference from co-running applications at a shared cache by using an auxiliary tag store. Then it models application slowdowns due to interference at both the shared cache and main memory and applies the model to improve performance and fairness of the applications. Iyer et al. [19] and Heracles [30] provide performance isolation by jointly partitioning both cache space and memory bandwidth. While

providing better end-to-end QoS than component-wise QoS approaches, their solutions are incomplete as they do not prevent blocking caused by shared DRAM request buffers. We show the existence of this problem and propose mVC to resolve it.

5.3 MiW on Recent Architectures

This study analyzed the MiW phenomenon in Intel Broadwell-based Xeon system and Skylake-based Xeon system where CAT was introduced. There is a study that analyzed the effect of CAT on Cascade Lake [70] and Ice Lake [69] based Xeon systems with more recent architecture [60].

In this study, a synthetic benchmark was designed and used in the experiment. The synthetic benchmark consists of a bandwidth benchmark that repeats a buffer of a given size several times until it is terminated [71]. Each iteration performs a load or store of every 64 bytes of data corresponding to the size of the cache line. Because there are no dependencies between successive requests, they can be done in parallel maximizing the load on main memory. Benchmarks estimate received bandwidth by measuring execution time and the number of memory operations completed. Depending on the size of the data buffer, you can make this benchmark either LLC sensitive or main memory sensitive.

To make the synthetic benchmark sensitive to main memory, the size of the buffer is set to 3 times the size of shared LLC, and through this, the impact of contention on the main memory resource can be confirmed. In order to make it sensitive to LLC, the buffer size was larger than the L2 size used in the experimental environment and smaller than the LLC size.

As a result of the experiment in the paper, it was shown that in Cascade Lake, when the number of interference cores increases, the LLC miss increases rapidly, resulting in memory bandwidth contention. In Ice Lake, LLC miss is constant even when the number of interference cores increases, but memory bandwidth contention still occurs. However, the cache miss is not null using CAT without contention. It is assumed that the reason why this phenomenon occurs is that Intel's address mapping applies the hash function over several bits to maintain a balance in the contiguous physical address space. The paper concludes that RDT management and monitoring do not always behave as expected.

Chapter 6

Conclusion

In this dissertation, we have demonstrated on real server machines how applications with more allocated LLC capacity can perform worse. Cache partitioning is promising for performance protection of a process by dedicating a portion of LLC, alleviating contention, and interference from other processes.

Because LLC is a shared resource with limited capacity, when we allocate more LLC capacity to one application, others receive relatively small LLC capacity. This results in a higher LLC MPKI and stresses the congested data path within memory controllers, which is another shared resource below the shared LLC, causing blocking, slowing down the entire system (a balloon effect). In particular, we identified this MiW phenomenon can impact

performance up to 39.5% on synthetic workloads. Also, latency-critical workloads could deteriorate 95th percentile latency as worse as 547% due to MiW.

To overcome this MiW, we proposed to virtualize the shared data path of memory controllers by mVCs. mVCs mostly eliminate the MiW phenomenon and improve the performance as the allocated LLC capacity increases, restoring the performance protection intended by cache partitioning. We also explored the design space of mVCs, changing the proportion of memory request queue and LLC capacity allocated to each mVC.

Finally, we can reduce the overall system cost using mVCs with a proper memory request queue size and LLC capacity while satisfying the target performance of latency-critical workloads even when executed with multiple workloads together. Results show that on SPEC CPU2006 workloads, up to 21.8% system cost can be saved while obtaining 90% of the performance compared to stand-alone execution on a dedicated machine. Note that this consolidation is only possible with mVC.

6.1 Discussion

So far, the impact of MiW and mVC on groups with high priority and groups with low priority has been analyzed, focusing on the inter-group. Intra-group, how to group applications is also an interesting topic.

In the case of an application group (group-A) with a high priority, many caches have already been allocated through the cache partition. Applications in the group operate like no-CP within the allocated cache, and in case of no-CP, MiW does not occur. If a higher priority application group (group-B) is created, group-B will be allocated more cache and group-A will be allocated less cache than before. In this case, group-A is allocated less cache and performance is reduced, but group-B with higher priority is not affected because of mVC. In addition, although group-A is allocated a small amount of cache, it is expected that MiW does not occur because cache partition is not applied inside group-A. Therefore, grouping applications with similar priorities will be the most efficient grouping method.

6.2 Future Work

Design space of mVC arbiter policy

In Chapter 4, we focused on four buffer allocation strategies for mVC. However, for the mVC arbiter, a simple round-robin manner arbitration logic was used. Further research of the design space of mVC arbiter policies can be a consideration since different policies can impact the performance. The mVC arbiter policy should be carefully considered. Wrong policy may affect DRAM commands and cause DRAM row buffer conflicts, which in turn may adversely affect overall system performance and even cause deadlocks or may not be fair.

Guideline for mVCs

In this paper, we showed that even with CP, we did not get the performance we expected for several benchmark applications. To solve this, we focused on showing that SLO can be satisfied by introducing mVC and combining appropriate knobs. Future work can aim to present guidelines that can provide appropriate values of knobs that can satisfy SLO to each application through real-time system monitoring. Through this, the user will be able to satisfy the SLO of the application by setting the knob values according to the guidelines.

Impact on virtual machines and containers

In this paper, we focused on general purpose benchmarks (SPEC) and real-time applications (TailBench). However, recently, deployment and services using VMs and containers for cloud services are becoming popular. It is also left as future work to check what impact there is in VMs and container environments with different application characteristics.

Bibliography

[1] S. Abousamra, A. El-Mahdy, and S. Selim, "Fair and Adaptive Online Set-based Cache Partitioning," in Computer Engineering & Systems (ICCES), International Conference on, 2011.

[2] J. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A Manycore Simulator with Application-level+ Simulation and Detailed Microarchitecture Modeling," in ISPASS, 2013.

[3] AMD, "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," 2006.

[4] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in ISCA, 2012.

[5] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2nd Edition," 2013.

[6] J. Chang and G. S. Sohi, Cooperative Caching for Chip Multiprocessors. ISCA, 2006.

[7] D. Chiou, P. Jain, S. Devadas, and L. Rudolph, “Dynamic Cache Partitioning via Columnization,” in DAC, 2000.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor,” IEEE Micro, vol. 30, no. 2, 2010.

[9] W. J. Dally and B. P. Towles, Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., 2003.

[10] J. Dean and L. A. Barroso, “The Tail at Scale,” Communications of the ACM, vol. 56, no. 2, 2013.

[11] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, “Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake,” IEEE Micro, vol. 37, no. 2, 2017.

[12] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems,” in ASPLOS, 2010.

[13] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, “Bandwidth Bandit: Quantitative Characterization of Memory Contention,” in CGO, 2013.

[14] B. Grot, S. W. Keckler, and O. Mutlu, “Preemptive Virtual Clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip,” in MICRO, 2009.

[15] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, “Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family,” in

HPCA, 2016.

[16] Intel, Intel Xeon Processor 7500 Series Datasheet, 2010.

[17] Intel, Intel 64 and IA-32 Architectures Software Developer's Manuals, 2018.

[18] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in ICS, 2004.

[19] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in SIGMETRICS, 2007.

[20] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications," in IISWC, 2016.

[21] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in ASPLOS, 2014.

[22] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in HPCA, 2010.

[23] M. K. Kumashikar, S. G. Bendi, S. Nimmagadda, A. J. Deka, and A. Agarwal, "14nm Broadwell Xeon processor family: Design methodologies and optimizations," in IEEE Asian Solid-State Circuits Conference (A-SSCC), 2017.

[24] J. W. Lee, "Globally Synchronized Frames for Guaranteed Quality-of-Service in Shared Memory Systems," Ph.D. dissertation, MIT, 2009.

[25] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-

Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," in ISCA, 2008.

[26] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs," in MICRO, 2010.

[27] F. Liu, X. Jiang, and Y. Solihin, "Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance," in HPCA, 2010.

[28] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Banklevel Interference in Multicore Systems," in PACT, 2012.

[29] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-Scale Latency-Critical Workloads," in ISCA, 2014.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in ISCA, 2015.

[31] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in ISCA, 2012.

[32] S. Mittal, "A Survey of Techniques for Cache Partitioning in Multicore Processors," ACM Computing Surveys (CSUR), vol. 50, no. 2, 2017.

[33] A. M. Molnos, M. J. Heijligers, S. D. Cotofana, and J. T. van Eijndhoven, "Compositional, Efficient Caches for a Chip Multi-processor," in DATE, 2006.

[34] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing

in On-chip Networks," in ISCA, 2009.

[35] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in ISCA, 2008.

[36] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair Queuing Memory Systems," in MICRO, 2006.

[37] J. Ouyang and Y. Xie, "LOFT: A High Performance Networkon- Chip Providing Quality-of-Service Support," in MICRO, 2010.

[38] M. Priyanka V P and M. K. Pramilarani, "An Analytical Model for Optimum Off-Chip Memory Bandwidth Partitioning in Multi-core Architectures," 2016.

[39] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in MICRO, 2006.

[40] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in ISCA, 2000.

[41] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in MICRO, 2010.

[42] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in ISCA, 2011.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in ASPLOS, 2002.

[44] W. Song, G. Kim, J. Chung, J. Ahn, J. W. Lee, and J. Kim, "History-based Arbitration for Fairness in Processor-Interconnect of NUMA Servers," in ASPLOS, 2017.

[45] Standard Performance Evaluation Corporation, "SPEC CPU2006," 2006. [Online]. Available: https://www.spec.org/cpu2006/

[46] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[47] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in MICRO, 2015.

[48] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in HPCA, 2013.

[49] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," Journal of Supercomputing, vol. 28, no. 1, 2004.

[50] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic Cache Allocation with Partial Sharing," in EuroSys, 2018.

[51] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-Service," in EuroSys, 2018.

[52] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in RTAS, 2014.

[53] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical

Page Coloring-based Multicore Cache Management," in EuroSys, 2009.

[54] M. Zhou, Y. Du, B. Childers, D. Mosse, and R. Melhem, "Symmetry-Agnostic Coordinated Management of the Memory Hierarchy in Multicore Systems," ACM TACO, 2016.

[55] J. Chung, Y. Ro, J. Kim, J. Ahn, J. Kim, J. Kim, J. W. Lee, and J. Ahn, "Enforcing Last-level Cache Partitioning through Memory Virtual Channels," in PACT, 2019

[56] Veitch, Paul, Edel Curley, and Tomasz Kantecki, "Performance Evaluation of Cache Allocation Technology for NFV Noisy Neighbor Mitigation," IEEE Conference on Network Softwarization, 2017.

[57] Intel, "Quiet Noisy Neighbor with Intel Resource Director Technology," 2017.

[58] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostic, "Make the Most Out of Last Level Cache in Intel Processors," in EuroSys, 2019.

[59] N. Guan, M. Lv, W. Yi, and G. Yu, "WCET Analysis with MRU Cache: Challenging LRU for Predictability," ACM Transactions on Embedded Computing Systems (TECS), 2014.

[60] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A Closer Look at Intel Resource Director Technology (RDT)," In Proceedings of the 30th International Conference on Real-Time Networks and Systems, 2022.

[61] J. L. Henning, "SPEC CPU2006 Memory Footprint," ACM

SIGARCH Computer Architecture News, 2006.

[62] D. Gove, “CPU2006 Working Set Size,” ACM SIGARCH Computer Architecture News, 2007.

[63] A. Yasin, “A Top-Down Method for Performance Analysis and Counters Architecture,” in ISPASS, 2014.

[64] Intel, “Intel VTuneTM Amplifier XE 2013.” [Online].

[65] A. Carvalho, “The New Linux ‘perf’ tools,” in linux kongress, 2010.

[66] R. H. Seethur Raviraj, “SPEC CPU2017: Performance, Energy and Event Characterization on Modern Processors,” The University of Alabama in Huntsville, 2018.

[67] S. Singh, and M. Awasthi, “Memory Centric Characterization and Analysis of Spec CPU2017 Suite,” in Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, 2019.

[68] A. Jaleel, “Memory characterization of workloads using instrumentation-driven simulation,” Web Copy: http://www. glue. umd. edu/ajaleel/workload, 2010.

[69] I. E. Papazian, “New 3rd Gen Intel Xeon Scalable Processor (Codename: Ice Lake-SP),” in Hot Chips Symposium, 2020.

[70] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, and S. Vora, “Cascade lake: Next generation intel xeon scalable processor,” in IEEE Micro, 2019.

[71] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms,” in RTAS, 2013.

국 문 초 록

최근 멀티코어 프로세서 기반 시스템은 학계 및 업계의 주목을 받고 있으며, 널리 사용되고 있다. 멀티코어 프로세서 기반 시스템은 서로 다른 특성을 가진 여러 응용 프로그램들이 동시에 실행되는데, 이 때 응용 프로그램들은 시스템의 여러 자원들을 공유하게 된다. 대표적인 공유 자원의 예로는 라스트 레벨 캐시(LLC) 및 메인 메모리를 들 수 있다. 이러한 단일 공유 메모리 시스템에서 서로 다른 특성을 가진 여러 응용 프로그램들 간에 공유 자원의 공정성을 보장하거나 특정 응용 프로그램이 다른 응용 프로그램으로부터 간섭을 받지 않도록 격리하는 것은 어려운 일이다.

이를 해결하기 위하여 최근 멀티코어 프로세서는 LLC 파티셔닝을 하드웨어적으로 제공하기 시작하였다. 사용자는 하드웨어적으로 제공된 LLC 파티셔닝을 통해 특정 응용 프로그램에 원하는 수준만큼 LLC를 할당하여 다른 응용 프로그램으로부터 간섭을 받지 않도록 격리할 수 있게 되었다. 일반적인 경우 LLC 용량을 많이 할당 받을수록 성능이 향상되는 경우가 많지만, 본 연구에서는 더 많은 LLC 용량을 할당 받은 응용 프로그램이 오히려 성능 저하된다는 사실(MiW, more is worse)을 하드웨어적 실험을 통해 확인하였다. 다양한 통제된 실험을 통해 LLC 파티셔닝을 통해 LLC 용량을 적게 할당 받은 응용 프로그램이 LLC 미스를 더 자주 발생시킨다는 사실을 확일 할 수 있었다. LLC 용량을 적게 할당 받은 응용 프로그램은 응용 프로그램들이 공유하는 메인 메모리 시스템에 스트레스를 가하고, LLC 파티셔닝을 통해 서로 격리를 하였음에도 불구하고 응용 프로그램의 성능을 저하시켰다.

MiW 현상을 해결하기 위해 본 연구에서는 메인 메모리 컨트롤러의 데이터 경로를 가상화하고 LLC 파티셔닝에 의해 그룹화된 각 응용 프로그램 그룹에 전용으로 할당되는 메모리 가상 채널(mVC)을 제안하였다. mVC를 통해 각 응용 프로그램 그룹은 독립적인 데이터 경로를 소유한 것처럼 가상화 된다. 따라서 특정 응용 프로그램 그룹이 데이터 경로를 독점하더라도 다른 응용 프로그램들은 성능 저하를 유발할 수 없게 되어 서로 격리된 환경을 조성한다. 추가적으로 mVC의 버퍼 크기를 조정하여 응용 프로그램 그룹의 성능 미세 조정이 가능하도록 하였다.

mVC를 도입함으로써 전체적인 시스템 비용을 줄일 수 있다. 지연 시간이 중요한 응용 프로그램과 처리량이 중요한 응용 프로그램을 함께 실행할 때 mVC가 없을 경우에는 지연 시간의 성능 기준치를 만족할 수 없었지만, mVC를 통해 성능 기준치를 만족하면서 시스템의 총 비용을 감소시킬 수 있었다. 멀티 칩 프로세서를 시뮬레이션한 실험 결과는 MiW 현상을 효과적으로 제거함을 보여주었다. 또한, 데이터 센터에서 응용 프로그램들의 동시 실행을 위한 추가적인 가능성을 제공하는 것을 보여주었다. 사례 연구를 통해 mVC를 도입하여 시스템 비용을 21.8%까지 절약할 수 있음을 보였으며, mVC를 도입하지 않은 경우에는 서비스 기준(SLO)을 만족하지 않음을 확인하였다.

주요어 : 라스트 레벨 캐시 파티셔닝, 주기억장치, 메모리 가상 채널, 공정성, QoS
학 번 : 2014-21730